End-to-End Incremental Data Integration via Knowledge Graphs

Javier Flores [a], Kashif Rabbani [b], Sergi Nadal [a], Cristina Gómez [a], Oscar Romero [a], Emmanuel Jamin [c] and Stamatia Dasiopoulou [c]

[a] *Department of Service and Information System Engineering, Universitat Politècnica de Catalunya, Barcelona, Spain. E-mails: jflores@essi.upc.edu, snadal@essi.upc.edu, cristina@essi.upc.edu, oromero@essi.upc.edu*
[b] *Department of Computer Science, Aalborg University, Aalborg, Denmark. E-mail: kashifrabbani@cs.aau.dk*
[c] *SEMBU, NTT Data, Barcelona, Spain. E-mails: emmanueljeanjacques.jamin@nttdata.com, stamatia.dasiopoulou@nttdata.com*

Abstract. Data integration, the task of providing a unified view over a set of data sources, is undoubtedly a major challenge for the knowledge graph community. Indeed, such flexible data structure allows to model the characteristics of source schemata, rich semantics for the global schema and the mappings between them. Yet, the design of such data integration systems still entails a manually arduous task. This becomes aggravated when dealing with heterogeneous and evolving data sources. To overcome these issues, we propose a fully-fledged semi-automatic and incremental data integration approach. By considering all tasks that compose the end-to-end data integration workflow (i.e., bootstrapping, schema matching, schema integration and generation of querying constructs), we are able to address them in a unified manner. We provide algorithms for each task, as well as theoretically prove the correctness of our approach and experimentally show its practical applicability.

Keywords: Data integration, Bootstrapping, Schema matching, Schema integration

1. Introduction

Nowadays, a plethora of data sources have been published through initiatives such as Open Data and Linked Open Data [1, 2]. Consequently, the volume of heterogeneous data has increased, thus leading to an increasing demand to integrate them in order to drive data-driven decision making [3]. Data Integration (DI), which is the task of creating a unified view over a set of heterogeneous data sources, becomes increasingly challenging in these settings [4]. Indeed, the design of effective DI systems becomes expensive and labor-intensive due to such massives amounts of data published adhering to different data models, schemata, or granularities [5]. As opposed to physical data integration, where data are warehoused into a common data repository, virtual data integration yields flexibility when aiming to handle a continuous data integration workflow. Precisely, an end-to-end virtual integration workflow (see Figure 1), involves three primary constructs (the source schemata, the global schema, and mappings). To that end, a set of chained tasks perform *1)* selection of data sources, *2)* extraction and transformation of schemas into a standard data model (bootstrapping), *3)* discovery and validation of overlapping concepts among schemata (schema matching), *4)* integration of schemata into a unified schema (schema integration), *5)* generation of DI constructs, and *6)* querying using rewriting algorithms. Over the decades, several proposals and systems have been developed to support one or more steps of such end-to-end workflow [6–9]. Nonetheless, there exist certain limitations that we outline as follows.

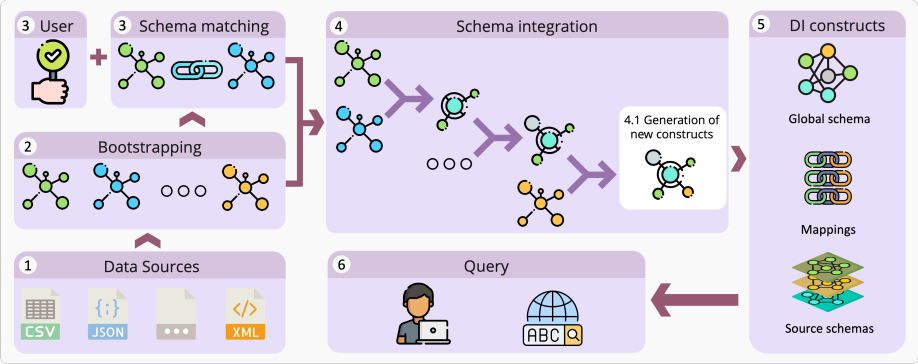

Fig. 1. Overview of a virtual end-to-end DI workflow

Bootstrapping heterogeneous data sources. Bootstrapping techniques are designed to work with specific data models focusing on well-defined schemas (e.g., relational data). Yet, most of the data available on the web comes in the form of semi-structured data sources (e.g., JSON, XML, and CSV). Approaches dealing with schemaless data sources require a provided schema, usually an ontology, to generate mappings (e.g., D2RQ, R2RML, or RML) that are used to materialize source instances into ontological facts [10, 11]. Nevertheless, it is unlikely that an ontology is complete with respect to the sources [12, 13]. Hence, maintaining the ontology and mappings requires a significant human effort [14]. We also encounter such limitation in approaches that exclusively bootstrap the source schemata (and not the instances). These also require high amount of manual labor and support only one data model [11, 15].

Need for incremental integration. DI is increasingly complex when dealing with numerous data sources that are usually generated within different application contexts and domains [16]. Such disparity entails a considerable effort to carefully find and validate overlapping concepts among data sources before integrating them [4]. The traditional DI process, which is performed in one shot, becomes rather limited as it requires the complete a priori identification of correspondences in schemata. An incremental DI approach is, hence, desired so that it reduces the complexity of integrating new data sources by reusing previously captured information and propagating the required changes to the DI constructs. To the best of our knowledge, Atom is the only solution implementing such incremental approach [17]. Yet its goal is the integration of taxonomies, thus omitting other key structural elements such as attributes.

Lack of unified management. There has been significant work to facilitate individual tasks of the DI workflow [15, 18, 19]. However, these have been developed in isolation, lacking a clear vision of the overall workflow. Consequently, effectively connecting each other, with the goal of providing an end-to-end workflow as in Figure 1, is an arduous task. Providing a single and unified framework is, thus, nowadays a major need in practice [20–22].

The limitations presented above have seriously hampered the development of an end-to-end DI workflow. The goal of this paper is, thus, to provide a single, coherent, all-encompassing approach for incremental and agnostic DI in a way that facilitates efficiently generating schemas of heterogeneous data sources and integrating them. Driven by the workflow in Figure 1, our approach presents the following features: *a)* extraction of schemata leveraging on the structure of schemaless data sources; *b)* standardization of such extracted schemata into a canonical data model (i.e., the RDFS graph data model) using the technique of production rules as in [23]; *c)* annotation-based schema integration for RDF graphs that allow to capture the relationships of the modeled data sources via unions and joins; *d)* automated derivation of the required DI constructs for specific querying systems (i.e., source schemata, schema mappings, and target schema). All such features are provided in such a way that they are agnostic of the target system, and are additionally performed in an incremental manner. As a result, the effort to maintain the integration process when new data sources appear is minimized.

Contributions. We summarize our contributions as follows:

- We present a general and agnostic algorithm to bootstrap schemaless data sources into RDF graphs representing their schema (supporting task 2 in Figure 1).

Approach	Bootstrapping type		Supported data sources			Implementation	
	Provided	Extracted	Data model	Input	Output	Availability	Automation
IncMap [24]	✓	✓	Relational	RDB Schema	Own declarative mappings	Unknown	Semi-automatic
D2RQ [25]		✓	Relational	RDB Schema	Own declarative mappings	Archieved	Automatic and manual
BootOX [26]	✓	✓	Relational	RDB Schema	R2RML	Unavailable	Automatic
Mirror [27]	✓		Relational	RDB Schema	R2RML	Active	Automatic
Janus [28] and XS2OWL [29]		✓	XML	XML Schema	Implicit in schema	Unknown	Automatic
DTD2OWL [30]		✓	XML	DTD Schema	Implicit in schema	Unknown	Automatic
J2RM [10]	✓		JSON	JSON file	J2RM	Active	Manual
Karma [9]		✓	Relational, CSV, XML, and JSON	Data source	R2RML	Active	Semi-automatic

Table 1
Comparison of bootstrapping state-of-the-art techniques

- We introduce an algorithm to incrementally capture the schema integration process guided by the output generated in the previous tasks (supporting task 4 in Figure 1). The generated graph is expressive enough to derive DI constructs for DI systems (supporting task 4.1 in Figure 1).
- An open source Java library, namely NextiaDI[1], implementing the abovementioned tasks. We showcase the effectiveness of NextiaDI when automatically generating the required constructs of a state-of-the-art DI system (supporting all tasks in Figure 1).

Outline. The rest of the paper is structured as follows. We discuss related work and introduce the formal background, respectively in Section 2. Next, in Section 3, we present the background of our approach to further dive in two main stages: data source bootstrapping (Section 4) and semi-automatic schema integration (Section 5). Our approach is extensively evaluated in Section 6. We finally conclude our paper and present future work in Section 7.

2. Related Work

Here, we present related work on supporting one or more tasks of the end-to-end DI workflow using graph data models. Due to the sparsity found in the literature, we have distinguish three categories: related work on bootstrapping, schema integration, and systems for end-to-end DI.

2.1. Related work on bootstrapping

Bootstrapping techniques play a critical role in standardizing and generating a source schema from data sources. There has been significant work in this area, as remarked in several surveys [11, 15, 31, 32]. In Table 1, we depict the most representative approaches. To better categorize them, we distinguish three dimensions: bootstrapping type, supported data sources, and implementation, which we detail as follows.

Bootstrapping type. We identify two main bootstrapping approaches. First, those that require an already defined schema (e.g., ontology) which is subsequently associated with the data sources through bootstrapping mappings (e.g., RML), and second, those that extract the schema from the physical structure of the available metadata (e.g., schemas, instances). The former requires detailed knowledge of the underlying mapping language, such as in [24], [26], and [10]. Using a provided ontology introduces a high degree of complexity when new independent data sources appear that are only partially represented in the reference ontology, or worse, are beyond its coverage [14]. This requires continuous involvement of domain and knowledge modeling experts to carefully extend and redesign

[1] https://www.essi.upc.edu/dtim/nextiadi/

Approach	Integration type	Strategy			Implementation	
		Alignments preservation	Source schema preservation	Incremental	Availability	Automation
PROMPT [35]	Full merge				Archieved	Semi-automatic
Chimaera [36] and FCA-Merge [37]	Full merge				Unknown	Semi-automatic
ONION [38]	Simple merge	✓	✓		Unknown	Semi-automatic
OntoMerge [39]	Simple merge	✓	✓		Unavailable	Automatic
ATOM [17]	Asymmetric merge			✓	Unknown	Semi-automatic
CoMerger [40]	Full merge	✓	✓		Active	Automatic
Chevalier et. al [41]	Full merge	✓	✓		Unknown	Semi-automatic

Table 2
Comparison of schema integration state-of-the-art techniques

the ontology. Moreover, a schema matching phase is required to define the mappings as in [26]. However, in [33] is illustrated that schema matching techniques tend to produce a high number of non-semantically correspondences due to data heterogeneity. Therefore, approaches falling in this category should efficiently support the maintenance of the schema and reduce the effort of schema matching. The second bootstrapping type defines rules to extract the schema from the available metadata as in [25], [28], [29], and [30], where the input is the data source to generate a graph-based schema using transformation rules. This type reduces the effort to maintain the schemas when new data sources appear since it only requires the application of transformation rules. However, these transformation rules are based on one data model and current approaches [9, 25, 30] cannot be extended to other data models, specially schemaless sources.

Supported data sources. Most approaches focus on bootstrapping relational data sources (e.g., [25] and [27]) with the goal of materializing data, and only few support semi-structured data models such as XML or JSON (e.g., [28] and [10]). Additionally, they lack a formal definition of how they address such task leading to ambiguous interpretations [22, 34]. Approaches such as [28], [29], and [30] introduce a dependence on well-defined schemata in sources such as XML. Consequently, this represents a limitation when a schema is not provided. Karma [9] supports schemaless data sources such as JSON, XML, and CSV by transforming them into a relational data model to apply the transformation rules and generate a graph-based schema to materialize data. However, using an intermediary model is not the best solution since important information related to the data source metamodel can be lost.

Implementation. When exploring and aiming to make use of the existing approaches, we found that most have no open implementation available. Only D2R [25], MIRROR [27], J2RM [10], and KARMA [9] can be found and adopted. Note, however, that D2R is no longer maintained.

2.2. Related work on schema integration

Schema integration is the process of generating a unified schema that represents a set of source schemata. This process requires semantic correspondences between their elements (i.e., alignments). In Table 2, we depict the most representative state-of-the-art approaches. We distinguish three dimensions: integration type, strategy, and implementation, which we detail as follows.

Integration type. Here, we adopt the distinction presented in [19], which categorizes the kinds of integration into three types, namely: simple merge, full merge, and asymmetric merge. The former integrates schemata by adding one-to-one bridge axioms between pairs of schemata. Approaches such as [38] and [39] preserve the original source schemata and alignments, which is a desirable property to manage source evolution [42]. However, their main drawback is a lack of common concepts which can lead to complex queries when a large number of schemas is integrated. The full merge integrates the schemas generating a new schema where the set of equivalent concepts is merged into a single new concept. In most approaches, the preservation of the original source schemas is lost. [40] is the only approach under this category able to preserve specific elements of the source schemas by request.

Approach	end-to-end DI workflow				Bootstrapping step		Schema integration		Implementation	
	Bootstrapping	Schema matching	Schema integration	Query	Type	Supported data sources	Type	Incremental	Availability	Last maintenance
Optique [43]	✓		✓	✓	Provided and extracted	Relational, sensor data and streaming	Simple merge	-	On request (demo)	2016
Mastro Studio [44]	✓			✓	Provided	Relational	-	-	Commercial use	Unknown
Squerall [45]	✓			✓	Provided	Relational, CSV, JSON and parquet	-	-	Open source	2021

Table 3

Comparison of end-to-end DI systems

Asymmetric merge can be performed using the simple and merge schema via one-to-many axioms to integrate schemas and derive a full merge. To the best of our knowledge, [17] is the only approach in this category. This kind of merge prioritizes one of the schemas (target) over the others (source), that is, the target schema will preserve all axioms and in case of disagreement with the target schema, elements of the source schema are removed from the integration.

Incremental integration. Most approaches (e.g., [35], [37], and OntoMerge [39]) integrate the source schemata in one shot. Thus, when a new data source has to be integrated the whole process is started from scratch without reusing the previously generated structures. Instead, [17] reuses the integrated structure to facilitate further integration incrementally. To that end, it adopts the asymmetric merge strategy, however, it is limited as it only supports the integration of taxonomies, that is, classes with no attributes and only hierarchical relations among them.

Implementation. All approaches are implemented, however only [40] is openly available. There exists a demonstration of [39] on its website[2], however, as of the date of this paper's submission the service is down. Regarding [35], it is available as a plugin to the Protégé tool, however it is not longer maintained. Overall, we also encountered that none of the tools offer any documentation for end-users.

2.3. Related work on end-to-end DI systems

The individual efforts in bootstrapping and schema integration have led to the generation of virtual end-to-end DI systems. These are prepared to perform each task systematically in a manually or semi-automatically way. Table 3 depicts the most representative systems. Note that DI systems such as Karma[9], Pool Party Semantic[8], or Ontotext[3] are excluded since their goal is to materialize data.

Optique. It is an end-to-end DI system developed as a result of research and industrial projects (e.g., [46]). It uses BootOX in the bootstrapping task to extract a schema or generate bootstrapping mappings to an existing ontology. Mappings are defined using the R2RML mapping language. Furthermore, the DI system allows the integration of relational, streaming, and sensor data, also providing a query interface. The reformulation of queries is performed by the Ontop tool [47]. The source code of the Optique platform is not publicly available.

Mastro studio. It is a virtual DI system over relational databases. It is currently available as commercial software maintained by OBDA Systems. The bootstrapping phase is performed by providing an ontology generating R2RML mappings. This system relies on its own native language to define mappings that describe a global schema in terms of local schemas for query rewriting. It is unclear if an integration phase is performed since data sources are connected to an existing ontology.

Squerall. Focuses on integrating heterogeneous data sources in data lakes. The bootstrapping phase extracts schema elements that are linked manually to semantic vocabularies for defining a graph schema. However, it is unclear if integration of schemas is performed. According to [49], it is possible to indicate which concepts should be integrated at query declaration. The implementation of the tool is published under Apache 2.0 license.

[2] http://cs-www.cs.yale.edu/homes/dvm/daml/ontology-translation.html

[3] https://www.ontotext.com/

2.4. Discussion

Above, we have presented an overview of the state-of-the art and most common limitations in bootstrapping, schema integration, and end-to-end DI systems. First, bootstrapping techniques rely mostly on provided schemata (or ontologies), which is challenging to maintain in continuous integration scenarios where data sources are from different domains. Instead, the automatic generation of such structures would be a more suitable approach. Yet, the support of schemaless data models is crucial in contemporary scenarios, however that is not commonly found in the related work. Regarding schema integration, the demand for continuous integration combined with the dynamic nature of data creates a need for incremental approaches. Additionally, these techniques must consider rich data structures, beyond taxonomies, including entities and properties. Finally, we advocate that systems aiming to provide the end-to-end DI workflow must strive for automation, thus reducing human intervention.

3. Preliminaries

In this section, we introduce the formal background of our approach and a running example which will be used in the following sections.

3.1. Formal background

3.1.1. Data source bootstrapping and production rules

Here, we present the formal definitions that are concerned with tasks 1 and 2 as depicted in Figure 1.

Datasets. Let $\mathbb{M}$ be the set of considered semi-structured and structured data models (e.g., CSV, JSON, or XML), then, a dataset $D_m = \{d_1, \ldots, d_n\}$, where $m \in \mathbb{M}$, is a collection of data elements where each d_i adheres to m's data model. We denote $schema(d)$ the set of elements that represent the physical structure of d's data model (e.g., the set of keys for a JSON document). In this paper, we assume all data elements in a dataset are *homogeneous* (i.e., they share the same schema), formally depicted as $\forall d_i, d_j \in D_m : schema(d_i) = schema(d_j)$, and hence with a slight abuse of notation we will refer to the schema of a dataset as $schema(D_m)$.

Typed graphs. We consider RDF graphs $G = (V_G, E_G)$, which are unweighted, directed edge-labeled graphs. As customary, we see E_G as a set of triples of the form $\langle s, p, o \rangle$, where p is a labeled edge from s to o with $s, o \in V_G$. Then, we say G is typed with respect to a graph $\mathcal{M} = (V_\mathcal{M}, E_\mathcal{M})$ (i.e., a metamodel), denoted as $G_\mathcal{M}$, if for every node $v \in V_G$ there exists a triple in E_G of the form $\langle v, \texttt{rdf:type}, m_i \rangle$, where $m_i \in V_\mathcal{M}$ (note we assume that V_G includes the elements of $V_\mathcal{M}$). We assume the existence of a set of boolean constraints $\mathcal{P}_\mathcal{M}$ for a given metamodel $\mathcal{M}$, which allow to guarantee that the elements of $G_\mathcal{M}$ are well-formed with respect to $\mathcal{M}$. Intuitively, $\mathcal{P}_\mathcal{M}$ concerns constraint checking aspects such as generalizations, referential integrity, cardinality, or keys. Then, we say $G_\mathcal{M}$ is *consistent* if all constraints in $\mathcal{P}_\mathcal{M}$ hold in $G_\mathcal{M}$.

Bootstrapping algorithm. A bootstrapping algorithm for a data model m is a function $\mathcal{B}_m : \mathbb{D}_m \rightarrow \mathbb{G}$ from the set $\mathbb{D}_m$ of all datasets adhering to m's data model to the set $\mathbb{G}$ of all graphs. Then, we say $\mathcal{B}_m$ is $\mathcal{M}$-*sound* if the graphs it generates are typed and consistent with respect to a metamodel $\mathcal{M}$. Formally, $\forall D_m \in \mathbb{D}_m$ we have that $\mathcal{B}_m(D_m)$ is typed with respect to $\mathcal{M}$, which we denote $G_\mathcal{M} = \mathcal{B}_m(D_m)$.

Graph queries. We consider the query language of *conjunctive graph queries* (CQs) as defined in [50]. Formally, a CQ Q is an expression of the form $Q(x_1, \ldots, x_m) \leftarrow \langle s_1, p_1, o_1 \rangle, \ldots, \langle s_n, p_n, o_n \rangle$, where each s_i, o_j are either vertex labels or variables, and each p_i is either an edge label or a variable. The semantics of answering a query CQ Q over a graph G (i.e., $Q_G(x_1, \ldots, x_m)$) is based on binding nodes and edges from G to the set of variables $\{x_1, \ldots, x_m\}$ that match the conjunction of patterns. In this paper, instead of the traditional homomorphism-based mapping semantics, where different variables can be bound to the same vertex, we assume the more restrictive isomorphism-based semantics, forcing mappings to be injective.

Production rules. Let $G_{\mathcal{S}}, G_{\mathcal{T}}$ be two typed graphs, respectively typed to $\mathcal{S}, \mathcal{T}$. Then, a source-to-target production rule p from $\mathcal{S}$ to $\mathcal{T}$ (i.e., $p_{\mathcal{S}\rightarrow\mathcal{T}}$) is an existential axiom of the form $\forall x \Theta_{G_{\mathcal{S}}}(x) \rightarrow \exists y \Psi_{G_{\mathcal{T}}}(x,y)$, where $\Theta_{G_{\mathcal{S}}}$ is a CQ over $G_{\mathcal{S}}$, and $\Psi_{G_{\mathcal{T}}}$ is a CQs over $G_{\mathcal{T}}$. A production system is a set of production rules $P = \{p_1, \ldots, p_k\}$, with an evaluation function $eval_P : \mathbb{G}_{\mathcal{S}} \rightarrow \mathbb{G}_{\mathcal{T}}$ from the set of all typed graphs with respect to $\mathcal{S}$ (i.e., $\mathbb{G}_{\mathcal{S}}$), to the set of all typed graphs with respect to $\mathcal{T}$ (i.e., $\mathbb{G}_{\mathcal{T}}$). We say a production system is *sound* if, after its evaluation, the resulting graph is typed with respect to $\mathcal{T}$. Formally, $\forall G_{\mathcal{S}} \in \mathbb{G}_{\mathcal{S}}$ we have that $eval_P(G_{\mathcal{S}})$ is typed with respect to $\mathcal{T}$. Likewise, we say a production system is *complete* if, after its evaluation, all candidate nodes in $G_{\mathcal{S}}$ are present in $G_{\mathcal{T}}$.

3.1.2. Integrating bootstrapped graphs and generating data integration systems

Here, we present the formal definitions that are concerned with tasks 3 and 4 as depicted in Figure 1.

Alignments. An alignment between two graphs G_A, G_B is a triple of the form $a = \langle v_a, v_b, \ell \rangle$, where v_a and v_b are nodes, respectively in V_{G_A} and V_{G_B}, and ℓ is a user-provided label for the aligned node. An alignment a is $\mathcal{M}$-compliant (i.e., compliant with the metamodel $\mathcal{M}$) if the aligned elements are typed to the same node in $\mathcal{M}$. Formally, $\exists \langle v_a, \texttt{rdf:type}, m \rangle \in E_{G_A}$ and $\exists \langle v_b, \texttt{rdf:type}, m \rangle \in E_{G_B}$, where $m \in V_{\mathcal{M}}$.

Graph integration algorithm. A graph integration algorithm is a function $\mathcal{I} : \overline{\mathcal{A}} \rightarrow \mathcal{G}$ from the set of all sets of alignments to the set of all graphs. We also consider the notion of $\mathcal{M}$-compliant graph integration algorithms, which entails that any graph generated by evaluating $\mathcal{I}$ is typed with respect to $\mathcal{M}$. This is formally defined as $\forall \mathcal{A} \in \overline{\mathcal{A}}$ we have that $\mathcal{I}(\mathcal{A})$ is typed with respect to $\mathcal{M}$.

Data integration systems and their generation. We follow Lenzerini's general framework for data integration systems [51]. Hence, a data integration system $\mathcal{K}$ is defined as a triple $\langle \mathcal{G}, \mathcal{S}, \varphi \rangle$, where $\mathcal{G}$ is the global schema, $\mathcal{S}$ the source schemata and φ the mappings between $\mathcal{S}$ and $\mathcal{G}$. Then, following the previous idea, a data integration system generation algorithm is a function $\mathcal{Q} : \mathbb{G} \rightarrow \mathbb{K}$ from the set of all graphs to the set of all data integration systems.

3.2. Running Example

Let us consider a data analyst interested in integrating two different sources about artworks from the Carnegie Museum of Art (CMOA)[4] and Cooper Hewitt Museum (Cooperhewitt)[5]. The former contains information such as the title, creator, creation and location for all artworks in the museum such as fine arts, decorative arts, photography, and contemporary art. The latter contains similar information about artworks exposed in the Cooper Hewitt museum such as painted architecture, decorative arts, sculpture and pottery. Figure 2 illustrates a fragment of the data example in JSON format. Note such snippets have been simplified for ease of illustration. Both data sources represent each artwork in a JSON file sharing similar concepts such as the title and url. Throughout the following sections, we will use this running example to showcase how our approach yields, from these semi-structured data sources, a data integration system leveraging on the formal definitions previously introduced.

4. Data source bootstrapping and production rules

In this section, we describe the bootstrapping algorithm to address task 2 in Figure 1. The algorithm takes as input a structured or semi-structured data source and produces a graph-based representation of its schema. This graph is typed with respect to the corresponding metamodel of the data source. Then, such typed graph is translated into a graph typed with respect to a canonical data model (in our case, RDFS) by applying a sound set of production rules. In the following, we present the metamodels required for our bootstrapping approach and the production rules.

4.1. Considered metamodels

As previously described, our approach is generic to any metamodel, as long as specific algorithms are implemented and shown to satisfy the soundness and completeness properties. In this paper, we showcase a specific in-

[4]https://github.com/cmoa/collection
[5]https://github.com/cooperhewitt/collection

Source: CMOA.json

```
{
    "title": "Keith Haring",
    "web_url": "http://collection.cmoa.org/
↪ CollectionDetail.aspx?item=1",
    "creator": [
      {
        "full_name": "Robert Mapplethorpe",
        "nationality": "American"
      }
    ]
  }
```

Source: Cooperhewitt.json

```
{
  "title": "Design for Portion of Stairway",
  "url":
↪ "collection.cooperhewitt.org/objects/68730511/",
  "description": "Drawing, Design for Portion of
↪ Stairway",
  "Participants": [
    {
      "person_name": "Charles Salagnad",
      "person_url":
↪ "collection.cooperhewitt.org/people/18062563/"
    }
  ]
}
```

Fig. 2. Running example excerpts.

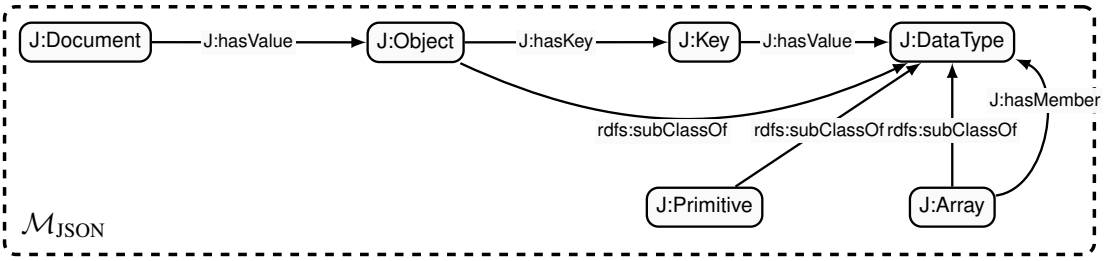

Fig. 3. Metamodel to represent graph-based schemata of JSON datasets (i.e., $\mathcal{M}_{\text{JSON}}$) inspired from [52]

stantiation of the framework considering the JSON data model. We, hence, present a metamodel for such semi-structured data model. Additionally, we adopt RDFS as target canonical metamodel for the production rules and the integration process considering its built-in ability to express and homogenize resources in a structured way distinguishing into three standardized layers (i.e., instances, model and meta-model).

4.1.1. Metamodels for semi-structured data sources

The first step of our approach requires the definition of a metamodel for each of the considered data models. We consider data models that are schemaless (i.e., with no fixed pre-defined schema for all their instances but with an implicit one), as well as those that have an explicit schema. As previously mentioned, we consider data elements in a semi-structured dataset to be homogeneous, hence we can consider that all data elements within a semi-structured dataset share the same implicit schema. In this paper, we showcase our approach on JSON data, to that end Figure 3, depicts the metamodel to represent graph-based JSON schemata (i.e., $\mathcal{M}_{\text{JSON}}$). Such meta-model (whose elements we prefix as $\mathtt{J}$) describes the basic constructs and the relationships between them. As can be seen, a J:Document consists of one root J:Object, which in turn contains at least one J:Key instances. Each J:Key is associated to one J:DataType value which is either a J:Primitive, a J:Array or a J:Object. We also assume elements of an J:Array to be homogeneous (e.g., it is not possible that an array combines numbers and strings), and thus it is composed of J:DataType elements. Furthermore, we consider three kinds of primitives, which are defined as its intances in any graph that is typed with respect to $\mathcal{M}_{\text{JSON}}$, these are J:Number, J:Boolean and J:String. In Appendix A, we present the complete set of constraints that guarantee that any typed graph with respect to $\mathcal{M}_{\text{JSON}}$ is consistent.

4.1.2. Canonical Metamodel

In our approach, in order to enable interoperability among graphs typed w.r.t. source-specific metamodels (e.g., JSON or CSV), we choose RDFS as canonical data model for the integration process. A significant advantage of RDFS is its built-in capabilities for meta-modeling, which supports different abstraction levels. Figure 4, depicts the fragment of the RDFS metamodel ($\mathcal{M}_{\text{RDFS}}$)[6] that we adopt. Considering that we aim to represent the schema

[6]https://www.omg.org/spec/ODM/1.1/PDF

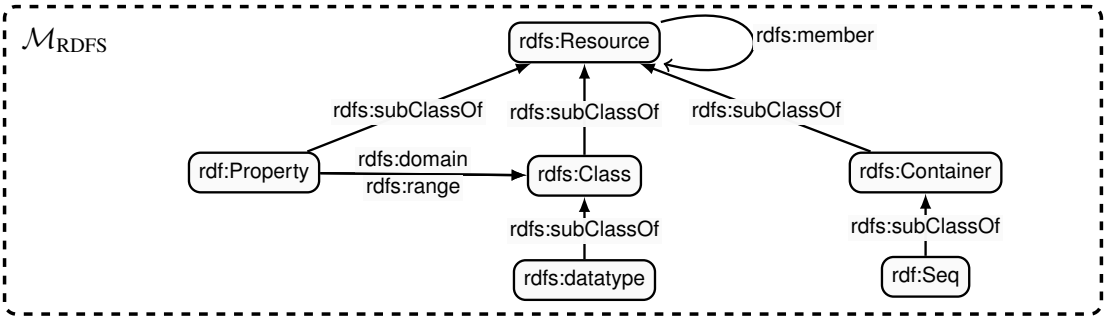

Fig. 4. Fragment of the RDFS metamodel (i.e., $\mathcal{M}_{\text{RDFS}}$) considered in this paper

of the underlying data sources, we only need to instantiate Class and their Property. Additionally, in order to model complex objects, we make use of Container resources, precisely Seq which allows the representation of arrays. As in the RDFS standard, we consider instances of the metamodel can be multi-type (i.e., an instance can contain multiple rdf:type edges to distinct elements of $\mathcal{M}_{\text{RDFS}}$). This is useful when defining instances of rdf:Seq, so they also become instances of rdfs:Class, allowing to relate via rdfs:range instances of both elements. In Appendix B, we present the complete set of constraints that guarantee that any typed graph with respect to $\mathcal{M}_{\text{RDFS}}$ is consistent.

4.2. Bootstrapping JSON data

Here, we present a bootstrapping algorithm to generate graph-based representations of JSON datasets. As depicted in Algorithm 1, the method DOCUMENT returns, for a given dataset, a graph typed with respect to the metamodel $\mathcal{M}_{\text{JSON}}$. To that end, it contains a set of functions, one per element of the metamodel, which have a shared signature consisting of *a)* the graph G where to populate the triples, and which is passed by reference; *b)* the JSON dataset or some of its children (e.g., embedded objects or arrays) D; and *c)* D's parent p (e.g., the key of an embedded object). For simplicity, we make use of a function IRI(s), that, given a string s, generates a unique IRI from s. Additionally, we use FRESH() to define fresh identifiers (i.e., synthetic strings that do not exist in G), and READFILE(D) to read from disk the content of a path D.

The goal of this algorithm is to instantiate the J:Document and J:Object elements using the document's root. Then, the method OBJECT recursively instantiates $\mathcal{M}_{\text{JSON}}$ with D's content. Precisely, for each key-value pair $\langle k, v\rangle$, which instantiate J:Key, we define its corresponding J:DataType vt and distinguish the cases of complex (i.e., objects and arrays) and simple (i.e., primitive) elements. Dealing with complex objects requires instantiating J:Object or J:Array with fresh IRIs (i.e., object or array identifiers). This is not the case for primitive elements, which are either connected to the three possible instances of J:Primitive (i.e., J:Number, J:Boolean, or J:String). The presence of such three possible instances in G is guaranteed by function INSTANTIATEMETAMODEL. We, additionally, annotate the elements contained in arrays. To that end, and assuming that elements in an array are homogeneous, we relate the instance of J:Array to an instance of J:DataType using the J:hasMember labeled edge.

Algorithm 1 Bootstrap a JSON dataset

Input: D is the name/path to the JSON dataset
Output: G is a typed graph with respect to $\mathcal{M}_{\text{JSON}}$ (i.e., $G_{\mathcal{M}_{\text{JSON}}}$)

1: **function** DOCUMENT(D)
2: $G \leftarrow \emptyset$
3: INSTANTIATEMETAMODEL(G)
4: $G \cup= \langle \text{IRI}(D), \texttt{rdf:type}, \texttt{J:Document} \rangle$
5: OBJECT(G, READFILE(D), D)
6: **return** G

1: **function** DATATYPE(G, D, p)
2: **if** ISOBJECT(D) **then** OBJECT(G, D, p)
3: **else if** ISARRAY(D) **then** ARRAY(G, D, p)
4: **else** PRIMITIVE(G, D, p)

1: **function** OBJECT(G, D, p)
2: $u' \leftarrow$ FRESH()
3: $G \cup= \langle \text{IRI}(u'), \texttt{rdf:type}, \texttt{J:Object} \rangle$
4: **for all** $(k, v) \in D$ **do**
5: $G \cup= \langle \text{IRI}(k), \texttt{rdf:type}, \texttt{J:Key} \rangle$
6: $G \cup= \langle \text{IRI}(u'), \texttt{J:hasKey}, \text{IRI}(k) \rangle \}$
7: DATATYPE(G, v, k)
8: $G \cup= \langle \text{IRI}(p), \texttt{J:hasValue}, \text{IRI}(u') \rangle$

1: **function** ARRAY(G, D, p)
2: $u' \leftarrow$ FRESH()
3: $G \cup= \langle \text{IRI}(u'), \texttt{rdf:type}, \texttt{J:Array} \rangle$
4: DATATYPE($G, D[0], u'$)
5: $G \cup= \langle \text{IRI}(p), \texttt{J:hasMember}, \text{IRI}(u') \rangle$

1: **function** PRIMITIVE(G, D, p)
2: **if** NUMBER(D) **then** $G \cup= \langle \text{IRI}(p), \texttt{J:hasValue}, \texttt{J:Number} \rangle$
3: **else if** BOOL(D) **then** $G \cup= \langle \text{IRI}(p), \texttt{J:hasValue}, \texttt{J:Boolean} \rangle$
4: **else** $G \cup= \langle \text{IRI}(p), \texttt{J:hasValue}, \texttt{J:String} \rangle$

1: **function** INSTANTIATEMETAMODEL(G)
2: $G \cup= \mathcal{M}_{\text{JSON}}$
3: $G \cup= \langle \texttt{J:Number}, \texttt{rdf:type}, \texttt{J:Primitive} \rangle$
4: $G \cup= \langle \texttt{J:Boolean}, \texttt{rdf:type}, \texttt{J:Primitive} \rangle$
5: $G \cup= \langle \texttt{J:String}, \texttt{rdf:type}, \texttt{J:Primitive} \rangle$

DOCUMENT(*CMOA.json*) { $\langle$*CMOA.json*, rdf:type, J:Document$\rangle$

INSTANTIATEMETAMODEL(G) { $\langle$J:String, rdf:type, J:Primitive$\rangle$
$\langle$J:Number, rdf:type, J:Primitive$\rangle$
$\langle$J:Boolean, rdf:type, J:Primitive$\rangle$

OBJECT(G, {...}, *CMOA.json*) { $\langle o_1$, rdf:type, J:Object$\rangle$
$\langle$*title*, rdf:type, J:Key$\rangle$
$\langle o_1$, J:hasKey, *title*$\rangle$
$\langle$*creator*, rdf:type, J:Key$\rangle$
$\langle o_1$, J:hasKey, *creator*$\rangle$
$\langle$*CMOA.json*, J:hasValue, $o_1\rangle$

PRIMITIVE(G, String, *title*) { $\langle$*title*, J:hasValue, J:String$\rangle$

ARRAY(G, [...], *creator*) { $\langle a_1$, rdf:type, J:Array$\rangle$
$\langle$*creator*, J:hasValue, $a_1\rangle$

OBJECT(G, {...}, a_1) { $\langle o_2$, rdf:type, J:Object$\rangle$
$\langle$*full_name*, rdf:type, J:Key$\rangle$
$\langle o_2$, J:hasKey, *full_name*$\rangle$
$\langle a_1$, J:hasMember, $o_2\rangle$

PRIMITIVE(G, String, *f_name*) { $\langle$*f_name*, J:hasValue, J:String$\rangle$

Fig. 5. Set of triples generated by Algorithm 1. In the left-hand side we depict the function that generates each corresponding set of triples

Example 1. *Retaking the running example introduced in Section 3.2, in Figure 5 we depict the set of triples that are generated by Algorithm 1 on a simplified version of the CMOA.json dataset. Note that, for the sake of simplicity, the keys* `web_url` *and* `nationality` *have been omitted.*

Proof of soundness. Here, we show that Algorithm 1 is $\mathcal{M}_{\text{JSON}}$-sound. This is, for any input JSON dataset, the graph it generates is typed with respect to $\mathcal{M}_{\text{JSON}}$ and consistent with respect to the set of boolean constraints $\mathcal{P}_{\mathcal{M}_{\text{JSON}}}$. By construction we can easily see that for every node $v \in V_G$ (where G is the output of Algorithm 1) there exists an edge $\langle v, \texttt{rdf:type}, m_i \rangle$ where $m_i \in V_{\mathcal{M}_{\text{JSON}}}$. Indeed, all instances of J:Document are declared in line 4 of function DOCUMENT, while all instances of J:Object and J:Key are declared, respectively, in lines 3 and 5 of function OBJECT. Instances of J:Array are declared in line 3 of function ARRAY. Regarding primitive data types, they are all declared once when instantiating the metamodel.

4.3. Production rules

In this subsection, we present the second step of our bootstrapping approach which is the translation of graphs typed with respect to a specific source data model (e.g., $\mathcal{M}_{\text{JSON}}$) into graphs typed with respect to a canonical metamodel (i.e., $\mathcal{M}_{\text{RDFS}}$). To that end, following the idea in [23], we present in the following the production system used to translate graphs typed with respect to JSON to RDFS. Precisely, the production system $P_{\text{JSON}\rightarrow\text{RDFS}}$ consists of eleven production rules which can be evaluated in no particular order. The remainder of this subsection is devoted to formally present the production rules that compose $P_{\text{JSON}\rightarrow\text{RDFS}}$.

Rule 1. Instances of J:Object are translated to instances of rdfs:Class.

$$\forall o\big(\langle o, \texttt{rdf:type}, \texttt{J:Object} \rangle(G)\big) \implies \exists c\big(\langle c, \texttt{rdf:type}, \texttt{rdfs:Class} \rangle(G') \wedge c = o\big) \qquad (R_1)$$

Rule 2. Instances of J:Array are translated to instances of rdfs:Class and rdf:Seq.

$$\forall a\big(\langle a, \texttt{rdf:type}, \texttt{J:Array}\rangle(G)\big) \implies \begin{array}{l}\exists s\big(\langle s, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle(G')\wedge \\ \langle s, \texttt{rdf:type}, \texttt{rdf:Seq}\rangle(G') \wedge s = a\big)\end{array} \tag{R_2}$$

Rule 3. Instances of J:Key are translated to instances of rdf:Property. Additionally, this requires defining the rdfs:domain of such newly defined instance of rdf:Property.

$$\forall o, k\big(\langle o, \texttt{J:hasKey}, k\rangle(G)\big) \implies \begin{array}{l}\exists c, p(\langle p, \texttt{rdf:type}, \texttt{rdf:Property}\rangle(G')\wedge \\ \langle p, \texttt{rdfs:domain}, c\rangle(G') \wedge p = k \wedge c = o)\end{array} \tag{R_3}$$

Rule 4. The rdfs:range of an instance of J:Primitive is its corresponding counterpart in the `xsd` vocabulary. Below we show the case for instances of J:String whose counterpart is xsd:string. The procedure for instances of J:Number and J:Boolean is similar using their pertaining type.

$$\begin{array}{l}\forall k, v\big(\langle k, \texttt{J:hasValue}, v\rangle(G)\wedge \\ \langle v, \texttt{rdf:type}, \texttt{J:String}\rangle(G)\big)\end{array} \implies \begin{array}{l}\exists p\big(\langle p, \texttt{rdf:type}, \texttt{rdf:Property}\rangle(G')\wedge \\ \langle p, \texttt{rdfs:range}, \texttt{xsd:string}\rangle(G') \wedge p = k\big)\end{array} \tag{R_4}$$

Rule 5. The rdfs:range of an instance of either J:Array or J:Object is the value itself.

$$\begin{array}{l}\forall k, v\big(\langle k, \texttt{J:hasValue}, v\rangle(G) \wedge (\\ \langle v, \texttt{rdf:type}, \texttt{J:Object}\rangle(G)\vee \\ \langle v, \texttt{rdf:type}, \texttt{J:Array}\rangle(G)))\end{array} \implies \begin{array}{l}\exists p, r\big(\langle p, \texttt{rdf:type}, \texttt{rdf:Property}\rangle(G')\wedge \\ \langle p, \texttt{rdfs:range}, r\rangle(G') \wedge p = k \wedge r = v\big)\end{array} \tag{R_5}$$

Rule 6. Instances of J:Primitive which are members of an instance of J:Array are connected to its corresponding counterpart in the `xsd` vocabulary using the rdfs:member property. We show the case for instances of J:String whose counterpart is xsd:string. The procedure for instances of J:Number and J:Boolean is similar using their pertaining type.

$$\begin{array}{l}\forall d, a\big(\langle a, \texttt{J:hasMember}, d\rangle(G)\wedge \\ \langle d, \texttt{rdf:type}, \texttt{J:String}\rangle(G)\big)\end{array} \implies \exists r\big(\langle \texttt{xsd:string}, \texttt{rdfs:member}, r\rangle(G) \wedge r = a\big) \tag{R_6}$$

Rule 7. Instances of J:Object or J:Array which are members of an instance of J:Array are connected via the rdfs:member property.

$$\begin{array}{l}\forall d, a\big(\langle a, \texttt{J:hasMember}, d\rangle(G) \wedge (\\ \langle d, \texttt{rdf:type}, \texttt{J:Object}\rangle(G)\vee \\ \langle d, \texttt{rdf:type}, \texttt{J:Array}\rangle(G)))\end{array} \implies \exists r, s\big(\langle s, \texttt{rdfs:member}, r\rangle(G) \wedge r = d \wedge s = a\big) \tag{R_7}$$

Example 2. *Here, we take as input the graph generated by our bootstrapping algorithm depicted in Figure 5. Then, Figure 6 shows the resulting graph typed w.r.t. the RDFs metamodel after applying the production rules. Each node and edge is annotated with the rule index that produced it.*

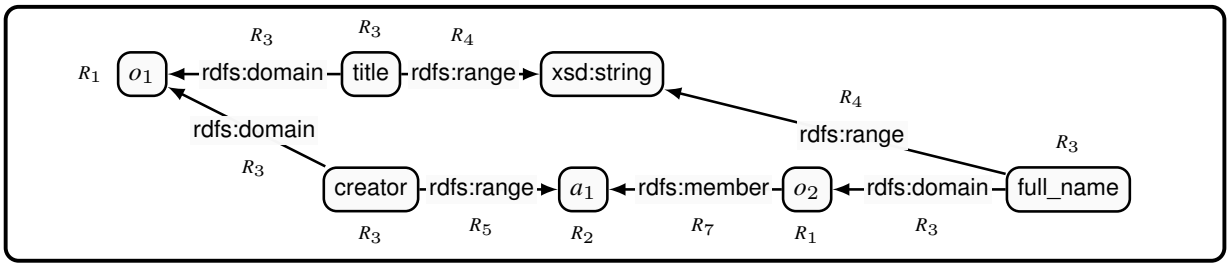

Fig. 6. Graph typed w.r.t. the RDFS metamodel resulting from evaluating the production rules

5. Incremental schema integration

This section describes the schema integration algorithm addressing task 4 in Figure 1. In our approach we extend the fragment of the RDFS metamodel previously discussed to include two new resources, namely :IntegratedResource and :JoinProperty (see Figure 7). In Appendix B.1, we present the complete set of constraint that guarantee that any integrated graph (IG) is consistent. These resources are essential to annotate how the underlying data sources they represent must be combined (e.g., union or join). The algorithm takes as input a pair of typed graphs and a list of semantic correspondences (i.e., alignments) between them. The latter are used to guide the integration algorithm when generating an IG, as well as a potential list of unused alignments. Note that we rely on existing schema matching techniques (e.g., LOGMAP[53]) to produce the alignments. To that end, the IG preserves the information from the sources and integrated resources, thus supporting their incremental construction. The list of unused alignments contains any alignment that is not integrated due to some conditions imposed by the integration algorithm, and it will be integrated once the conditions are fulfilled in further executions. In the following, we present the formal foundations for generating IGs, and we propose an algorithm precisely focused for the case of classes and properties.

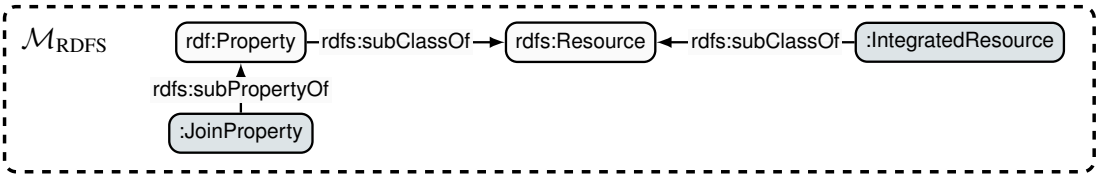

Fig. 7. Extension of the rdfs metamodel for the integration annotation

Example 3. *Continuing the running example, Figure 8 illustrates the result of the bootstrapping step to generate the typed graph representation for the data sources CMOA.json and Cooperhewitt.json. Note that the range properties are omitted due to space reasons. For ease of illustration, we distinguish the property into* Datatype property *and* Object property*. In practice, we use* rdf:Property *and distinguish into these two types by checking their range property. Therefore, we use the color information present in Figure 8 to represent* rdfs:Class*,* Datatype property *and* Object property*. We will use dark color to depict integrated resources as the following* :IntegratedResource *of type class,* :IntegratedResource *of type data type property and* :IntegratedResource *of type object property.*

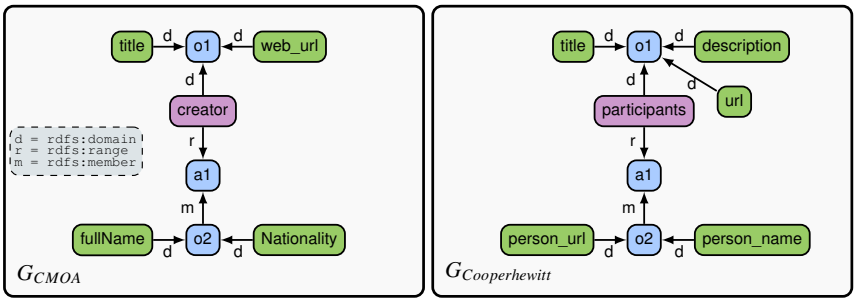

Fig. 8. Two extracted canonical RDF representations from CMOA and Cooperhewitt sources

5.1. Integration of resources

In this paper, we present an agnostic and incremental process (i.e., a graph integration algorithm as formally defined in Section 3.1.2) to integrate schemas by generating an integrated graph G_I that contains the source graph-based schemas and annotations for the integration process. The generation of G_I requires two typed graphs G_A and G_B and a list of alignments a of the form $\langle R_A, R_B, l\rangle$ where R_A is a URI resource in G_A and R_B is a URI resource in G_B, R_A and R_B pertain to the same type (e.g., class aligned with class), and l is a user-provided label for the aligned resource. Then, we generate an integrated graph G_I' as the union of G_A and G_B and use the set of alignments to

generate the corresponding integrated resources and their metadata. In the following, we present the invariants given G_I that guarantee the creation and propagation of the integrated resources in G'_I . Note that for explainability, we only present the invariants for class resources. However, the procedure for properties similarly requires the use of the proper axioms such as rdfs:subPropertyOf instead of rdfs:subClassOf.

I1. If R_A and R_B are not :IntegratedResource, then R_A and R_B are sub-class of a new :IntegratedResource R_l defined as the URI representation of l (i.e., $R_l = generateURI(l)$). Formally:

$$\begin{array}{l}\forall\langle R_A, R_B, l\rangle \in A_R : \\ \langle R_A, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_I \wedge \\ \langle R_B, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_I \wedge \\ \langle R_A, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_I \wedge \\ \langle R_B, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_I\end{array} \implies \begin{array}{l}\exists R_l = generateURI(l) : \\ \langle R_l, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_{I'} \wedge \\ \langle R_l, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_{I'} \wedge \\ \langle R_A, \texttt{rdfs:subClassOf}, R_l\rangle \in G_{I'} \wedge \\ \langle R_B, \texttt{rdfs:subClassOf}, R_l\rangle \in G_{I'}\end{array}$$

I2. If R_A is not an :IntegratedResource and R_B is an :IntegratedResource, then R_A is a sub-class of R_B. Formally:

$$\begin{array}{l}\forall\langle R_A, R_B, l\rangle \in A_R : \\ \langle R_A, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_I \wedge \\ \langle R_B, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_I \wedge \\ \langle R_A, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_I \wedge \\ \langle R_B, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_I\end{array} \implies \begin{array}{l}\langle R_A, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_{I'} \wedge \\ \langle R_B, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_{I'} \wedge \\ \langle R_B, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_{I'} \wedge \\ \langle R_A, \texttt{rdfs:subClassOf}, R_B\rangle \in G_{I'}\end{array}$$

I3. If R_A is an :IntegratedResource and R_B is not an :IntegratedResource, then R_B is a sub-class of R_A. Formally:

$$\begin{array}{l}\forall\langle R_A, R_B, l\rangle \in A_c : \\ \langle R_A, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_I \wedge \\ \langle R_B, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_I \wedge \\ \langle R_A, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_I \wedge \\ \langle R_B, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_I\end{array} \implies \begin{array}{l}\langle R_A, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_{I'} \wedge \\ \langle R_A, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_{I'} \wedge \\ \langle R_B, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_{I'} \wedge \\ \langle R_B, \texttt{rdfs:subClassOf}, R_A\rangle \in G_{I'}\end{array}$$

I4. If R_A and R_B are :IntegratedResource, then the new integrated resource C_l defined as the URI representation of l (i.e., $C_l = generateURI(l)$) replaces R_A and R_B. Formally:

$$\begin{array}{l}\forall\langle R_A, R_B, l\rangle \in A_c : \\ \langle R_A, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_I \wedge \\ \langle R_B, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_I \wedge \\ \langle R_A, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_I \wedge \\ \langle R_B, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_I \wedge \\ \langle r, \texttt{rdfs:subClassOf}, R_A\rangle \in G_I \wedge \\ \langle r', \texttt{rdfs:subClassOf}, R_B\rangle \in G_I \wedge r \neq r'\end{array} \implies \begin{array}{l}\exists R_l = generateURI(l) \wedge \forall r, r' : \\ \langle R_l, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_{I'} \wedge \\ \langle R_l, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_{I'} \wedge \\ \langle R_A, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_{I'} \wedge \\ \langle R_A, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \notin G_{I'} \wedge \\ \langle R_B, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_{I'} \wedge \\ \langle R_B, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \notin G_{I'} \wedge \\ \langle r, \texttt{rdfs:subClassOf}, R_A\rangle \notin G_{I'} \wedge \\ \langle r', \texttt{rdfs:subClassOf}, R_B\rangle \notin G_{I'} \wedge \\ \langle r, \texttt{rdfs:subClassOf}, C_l\rangle \in G_{I'} \wedge \\ \langle r', \texttt{rdfs:subClassOf}, C_l\rangle \in G_{I'}\end{array}$$

Algorithm 2 implements each invariant to generate the corresponding integrated metadata and support incremental construction and propagation of the metadata. This algorithm is the core for integrating any resource. Note that invariant *I1* creates a new :IntegratedResource of type class, invariants *I2* and *I3* reuse an existing :IntegratedResource and invariant *I4* replace an :IntegratedResource. To acomplish *I4*, our algorithm uses the method *replaceIntegratedResource* depicted in Algorithm 3. This algorithm offers the basic functionality to replace the resources that were integrated by an :IntegratedResource$_{old}$ with :IntegratedResource$_{new}$. Note that depending on the resource, it is required to manage all resources (e.g., properties) connected to the :IntegratedResource$_{old}$ to connect them to the :IntegratedResource$_{new}$. However, this depends on the type of resource (e.g., class or property). For example, replacing an :IntegratedResource$_{old}$ of type class requires updating all properties referencing :IntegratedResource$_{old}$ by rdfs:range or rdfs:domain to reference :IntegratedResource$_{new}$.

Therefore, we introduce the method concordanceRelations to redirect the relations from an old resource to a new resource. This method is polymorphic since the updates will depend on the specific needs of the resource type.

Algorithm 2 Integration of resources

Input: Two typed graphs G_A and G_B with a set of class alignments $A = \{a_1, ..., a_n\}$ and a set of unused alignments A_{unused} from previous integrations (For the first iteration, this parameter is an empty set)
Output: Generates the integrated graph $G_{I'}$ and the set of alignments not used $A_{unused'}$

```
function INTEGRATERESOURCES(G_A, G_B, A, A_unused)
    G_I ← G_A ∪ G_B
    G_I' ← G_I
    A_unused' = A_unused
    for all ⟨R_A, R_B, l⟩ in A do
        R_l ← generateURI(l)
        if any condition then
            if ⟨R_A, rdf:type, :IntegratedResource⟩ ∈ G_I' ∧ ⟨R_B, rdf:type, :IntegratedResource⟩ ∈ G_I' then        ▹ Invariant I4
                G_I' \= replaceIntegratedResource(G_I', R_A, R_l)                                                    ▹ see Algorithm 3
                G_I' \= replaceIntegratedResource(G_I', R_B, R_l)
            else if ⟨R_A, rdf:type, :IntegratedResource⟩ ∈ G_I' then                                                  ▹ Invariant I3
                G_I' ∪= ⟨R_B, rdfs:subClassOf, R_A⟩
            else if ⟨R_B, rdf:type, :IntegratedResource⟩ ∈ G_I' then                                                  ▹ Invariant I2
                G_I' ∪= ⟨R_A, rdfs:subClassOf, R_B⟩
            else                                                                                                      ▹ Invariant I1
                G_I' ∪= ⟨R_l, rdf:type, :IntegratedResource⟩
                G_I' ∪= ⟨R_l, rdf:type, rdfs:Class⟩
                G_I' ∪= ⟨R_A, rdfs:subClassOf, R_l⟩
                G_I' ∪= ⟨R_B, rdfs:subClassOf, R_l⟩
        else
            A_unused' ∪= (R_A, R_B, l)
    return ⟨G_I, A_unused'⟩
```

Algorithm 3 Replace integrated resource

Input: G_I is the integrated graph, R_{old} is the old integrated resource to be replaced , R_{new} is the new integrated resource
Output: Generates the integrated graph $G_{I'}$ containing the new triples with R_{new}

```
function REPLACEINTEGRATEDRESOURCE(G_I, R_old, R_new)
    G_I' ← G_I
    G_I' ∪= ⟨R_new, rdf:type, :IntegratedResource⟩
    G_I' \= ⟨R_old, rdf:type, :IntegratedResource⟩
    for all c in ⟨?r, rdfs:subClassOf, R_old⟩(G_I') do
        G_I' \= ⟨r, rdfs:subClassOf, R_old⟩
        G_I' ∪= ⟨r, rdfs:subClassOf, R_new⟩
        G_I' ∪= concordanceRelations(G_I', R_old, R_new)
    return G_I'
```

The abovementioned invariants are the core to perform schema integration as depicted in Algorithm 4, which is the primary integration algorithm to create an IG. The algorihm expects sets of alignments to integrate resources pertaining to the same type and implements the methods *IntegrateClasses* for class resources, *IntegrateDataTypeProperties* for data type properties and *IntegrateObjectProperties* for object properties based on all invariants of the Algorithm 2. Notice that the algorithm relies on the discovery result of any alignment tool (e.g. LOGMAP) and to prevent and reduce the integration of non-semantically resources, the algorithm follows a class oriented integration, that is, properties can only be integrated once their corresponding domain or range classes are integrated into the same integration class. Thus, this integration ensures that properties are semantically related and considered as a union operation. Moreover, properties that belong to weak entities such as latitude and longitude can be integrated with a post-process task. This integration represents a join operation between two entities not semantically related. In the following subsections, we depict how integration is performed over classes and properties along with the Example 3 to illustrate the integration process.

5.1.1. Integration of classes

Algorithm 4 Incremental global graph integration

Input: Two typed graphs G_A and G_B with a set of class alignments A_C, data type property alignments A_{DTP}, object property alignments A_{OP} and any property alignments A_{unused} not used in previous integrations (For the first iteration, this parameter is an empty set)
Output: Generates the integrated graph G_I and a set of property alignments $A_{unused'}$

function $\mathcal{I}(G_A, G_B, A_C, A_{DTP}, A_{OP}, A_{unused})$
  $G_I \leftarrow G_A \cup G_B$
  $\langle G_I, A_{unused'} \rangle = IntegrateClasses(G_I, A_C, A_{unused})$
  $\langle G_I, A_{unused'} \rangle = IntegrateDataTypeProperties(G_I, A_{DTP}, A_{unused'})$
  $\langle G_I, A_{unused'} \rangle = IntegrateObjectProperties(G_I, A_{OP}, A_{unused'})$
  return $\langle G_I, A_{unused'} \rangle$

Algorithm 5 ConcordanceRelations – Classes

Input: G_I is the integrated graph, C_{old} is the old integrated class to be replaced, C_{new} is the new integrated class
Output: Generates the integrated graph $G_{I'}$ containing the new triples with C_{new}

function CONCORDANCERELATIONS(G_I, C_{old}, C_{new})
  for all p in $\langle ?p, \texttt{rdfs:domain}, C_{old} \rangle (G_{I'})$ **do**
    $G_{I'} \setminus= \langle p, \texttt{rdfs:domain}, C_{old} \rangle$
    $G_{I'} \cup= \langle p, \texttt{rdfs:domain}, C_{new} \rangle$
  for all p in $\langle ?p, \texttt{rdfs:range}, C_{old} \rangle (G_{I'})$ **do**
    $G_{I'} \setminus= \langle p, \texttt{rdfs:range}, C_{old} \rangle$
    $G_{I'} \cup= \langle p, \texttt{rdfs:range}, C_{new} \rangle$
  return $G_{I'}$

The class integration uses the method *IntegrateClasses* implementing all invariants presented in Algorithm 2. This integration type does not require any condition. Note that for invariant $I4$, we must manage the axioms when an :IntegratedResource of type class is replaced. The method *concordanceRelations* in Algorithm 5 takes as input two rdfs:Class and the IG to ensure that all rdf:Property axioms that are connected to :IntegratedResource$_{old}$ (C_{old}) (e.g., rdfs:domain or rdfs:range) will reference the :IntegratedResource$_{new}$ (C_{new}) in the generated IG ($G_{I'}$). Moreover, as mentioned earlier our approach is class oriented and alignments not used in previous integrations are saved in the set A_{unused}. Then, after performing the integration of a pair of classes is recommended to perform an additional step to revise the set A_{unused} that could fulfill the conditions imposed by the properties integration (see section 5.1.2).

Example 4. *Retaking the running example, consider the alignments depicted in Table 4 between G_{CMOA} and $G_{Cooperhewitt}$. Let us consider the alignment* o1 *and* o1*. For this case, invariant I1 is applied as illustrated in Figure 9. Thus, both classes are connected to a newly defined instance of class* :IntegratedResource *and* rdfs:Class*, namely* Artworks*. For this example, two* :IntegratedResource *were generated:* Artworks *and* Creator*. Note, the set A_{unused} is empty since this is the first integration performed.*

G_{CMOA}	$G_{Cooperhewitt}$	user-provided label l
o1	o1	Artworks
title	title	Title
web_url	url	URL
o2	o2	Creator
fullName	personName	Name

Table 4
Discovered correspondences for G_{CMOA} and $G_{Cooperhewitt}$

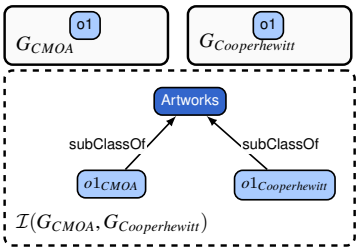

Fig. 9. Integration of two classes from G_{CMOA} and $G_{Cooperhewitt}$

5.1.2. Integration of properties

The integration of properties distinguishes three cases: Datatype property, Object property and :JoinProperty. Those cases requires different conditions to allow integrating properties. For Datatype property, we only integrate them if they are part of the same entity, that is, if their corresponding domain rdfs:Class are already integrated into the same :IntegratedResource of type class. Therefore, we include these conditions to all invariants in Algorithm 2 for data type properties. In the following, we present invariant I1 with the respective conditions. Note that for the remaining invariants, the conditions are similar.

$$
\begin{array}{l}
\forall \langle R_A, R_B, l\rangle \in A_R : \\
\langle R_A, \texttt{rdfs:domain}, C_A\rangle \in G_I \wedge \langle C_A, \texttt{rdfs:subClassOf}, C_{IA}\rangle \in G_I \wedge \\
\langle R_B, \texttt{rdfs:domain}, C_B\rangle \in G_I \wedge \langle C_B, \texttt{rdfs:subClassOf}, C_{IB}\rangle \in G_I \wedge \\
\langle C_{IA}, \texttt{rdf:type}, \texttt{IntegratedResource}\rangle \in G_I \wedge C_{IA} = C_{IB} \wedge \\
\langle R_A, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_I \wedge \\
\langle R_B, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_I \wedge \\
\langle R_A, \texttt{rdf:type}, \texttt{rdf:Property}\rangle \in G_I \wedge \\
\langle R_B, \texttt{rdf:type}, \texttt{rdf:Property}\rangle \in G_I
\end{array}
\implies
\begin{array}{l}
\exists R_l = generateURI(l) : \\
\langle R_l, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_{I'} \wedge \\
\langle R_l, \texttt{rdf:type}, \texttt{rdf:Property}\rangle \in G_{I'} \wedge \\
\langle R_A, \texttt{rdfs:subPropertyOf}, R_l\rangle \in G_{I'} \wedge \\
\langle R_B, \texttt{rdfs:subPropertyOf}, R_l\rangle \in G_{I'}
\end{array}
$$

Algorithm 6 Integration of Data type properties — adapted from Algorithm 2

7: $C_{domainA} \leftarrow GetIntegratedClass(G_{I'}, R_A, \texttt{rdfs:domain})$
8: $C_{domainB} \leftarrow GetIntegratedClass(G_{I'}, R_B, \texttt{rdfs:domain})$
9: **if** $\langle C_{domainA}, \texttt{rdf:type}, : IntegrationClass\rangle \in G_{I'} \wedge C_{domainA} = C_{domainB}$ **then**
10: ... ▹ Invariants I4, I3, I2
16: ... ▹ Start of Invariant I1
17: $C_{rangeA} \leftarrow GetIntegratedClass(G_{I'}, R_A, \texttt{rdfs:range})$
18: $G_{I'} \cup= \langle R_l, \texttt{rdfs:range}, C_{rangeA}\rangle$
19: $G_{I'} \cup= \langle R_l, \texttt{rdfs:domain}, C_{domainA}\rangle$
20: ... ▹ End of Invariant I1
21: **else**
22: $A_{unused'} \cup= (R_A, R_B, l)$
23: return $\langle G_{I'}, A_{unused'}\rangle$

Algorithm 7 ConcordanceRelations — Properties

Input: G_I is the integrated graph, P_{old} is the old integrated property to be replaced, P_{new} is the new integrated property
Output: Generates the integrated graph $G_{I'}$ containing the new triples with P_{new}

1: **function** CONCORDANCERELATIONS(G_I, P_{old}, P_{new})
2: **for all** c in $\langle P_{old}, \texttt{rdfs:domain}, ?c\rangle(G_{I'})$ **do**
3: $G_{I'} \setminus= \langle P_{old}, \texttt{rdfs:domain}, c\rangle$
4: $G_{I'} \cup= \langle P_{new}, \texttt{rdfs:domain}, c\rangle$
5: ▹ The following loop only applies to object properties
6: **for all** c in $\langle P_{old}, \texttt{rdfs:range}, ?c\rangle(G_{I'})$ **do**
7: $G_{I'} \setminus= \langle P_{old}, \texttt{rdfs:range}, c\rangle$
8: $G_{I'} \cup= \langle P_{new}, \texttt{rdfs:range}, c\rangle$,
9: return $G_{I'}$

Algorithm 6 represents the *IntegrateDataTypeProperties* method in Algorithm 4. The implementation of *IntegrateDataTypeProperties* method is similar to Algorithm 2 except for the additional conditions and the manipulation of the domain and range axioms. Note that if the conditions are not fulfilled, the algorithm does not integrate the alignment and only saves the alignment in the set of A_{unused} until their domain is integrated into the same :IntegratedResource in further integrations. Note that this will depend on the validated result of the schema matching approaches. For the manipulation of the axioms, the domain of the :IntegratedResource of type property is an :IntegratedResource of type class. And for the range we assign the more flexible xsd type (e.g., xsd:string). For the integration of object properties we follow a similar approach, we only integrate object properties if their domain and range are already integrated. Then, we add the proper conditions to all invariants presented in Algorithm 2. We present the conditions for invariant I1 for object properties in the following.

$$
\begin{array}{l}
\forall \langle R_A, R_B, l\rangle \in A_R : \\
\langle R_A, \texttt{rdfs:domain}, C_{AD}\rangle \in G_I \wedge \langle C_{AD}, \texttt{rdfs:subClassOf}, C_{IAD}\rangle \in G_I \wedge \\
\langle R_B, \texttt{rdfs:domain}, C_{BD}\rangle \in G_I \wedge \langle C_{BD}, \texttt{rdfs:subClassOf}, C_{IBD}\rangle \in G_I \wedge \\
\langle C_{IAD}, \texttt{rdf:type}, \texttt{IntegratedResource}\rangle \in G_I \wedge C_{IAD} = C_{IBD} \wedge \\
\langle R_A, \texttt{rdfs:range}, C_{AR}\rangle \in G_I \wedge \langle C_{AR}, \texttt{rdfs:subClassOf}, C_{IAR}\rangle \in G_I \wedge \\
\langle R_B, \texttt{rdfs:range}, C_{BR}\rangle \in G_I \wedge \langle C_{BR}, \texttt{rdfs:subClassOf}, C_{IBR}\rangle \in G_I \wedge \\
\langle C_{IAR}, \texttt{rdf:type}, \texttt{IntegratedResource}\rangle \in G_I \wedge C_{IAR} = C_{IBR} \wedge \\
\langle R_A, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_I \wedge \\
\langle R_B, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_I \wedge \\
\langle R_A, \texttt{rdf:type}, \texttt{rdf:Property}\rangle \in G_I \wedge \\
\langle R_B, \texttt{rdf:type}, \texttt{rdf:Property}\rangle \in G_I
\end{array}
\implies
\begin{array}{l}
\exists R_l = generateURI(l) : \\
\langle R_l, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_{I'} \wedge \\
\langle R_l, \texttt{rdf:type}, \texttt{rdf:Property}\rangle \in G_{I'} \wedge \\
\langle R_A, \texttt{rdfs:subPropertyOf}, R_l\rangle \in G_{I'} \wedge \\
\langle R_B, \texttt{rdfs:subPropertyOf}, R_l\rangle \in G_{I'}
\end{array}
$$

For the implementation of the Object property integration is very similar to Algorithm 6. The algorithm only requires the proper conditions and the manipulation of the axioms as we performed for Datatype property. We illustrate the integration of properties in Example 5.

Example 5. *Once the class integration is performed, we proceed to integrate* Datatype property*. Continuing the Example 4, the algorithm will integrate all* Datatype property *using invariant I1 since their domains were already integrated. In case the domains will not be already integrated, the properties will be saved in the set of unused alignments A_{unused}. Figure 10 illustrates the integration process of integrating* web_url *and* url*. The integration algorithm uses*

rdfs:subPropertyOf to connect both Datatype property *to the* :IntegratedResource *of type property* URL*. For this example, the following* :IntegratedResource *were generated:* URL*,* Title *and* Name*. Finally, Figure 11 depicts the complete integrated graph generated from Example 4 and 5. The result of this DI generates 1* :IntegratedResource *of type class and 3* :IntegratedResource *of type property. Note that the set A_{unused} contains the alignments not used during this integration giving the opportunity to use them in further integrations.*

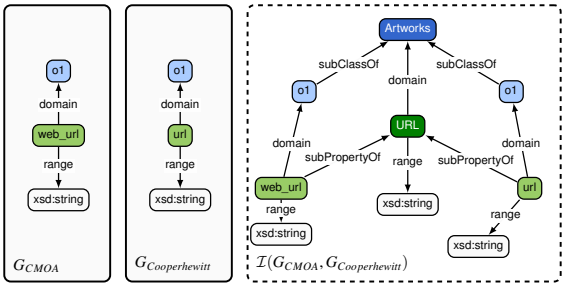

Fig. 10. Result of integrating two classes and two data type properties from G_{CMOA} and $G_{CooperhewittInt}$

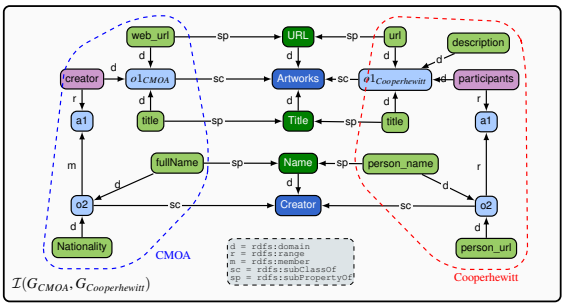

Fig. 11. Final Integration graph from Examples 4 and 5

Finally, the integration of Datatype property without conditions is a unique case, namely :JoinProperty, and should be performed in a post-process task. This type of integration is performed when the domains are not semantically related but the properties are semantically related. We can consider this case as a join operation. Algorithm 8 depicts this integration type. Note that this is performed on a user request. Having no conditions allow us to integrate properties from completly different entities. To this aim, the integration of the data type properties considers all invariants of the core Algorithm 2 to create an :IntegratedResource of type property. Moreover, we express the join relationship by creating an object property that connects both domain rdfs:Class of the Datatype property. This object property aims to add semantic meaning to the relation of the properties domain. This Object property is connected to the :IntegratedResource using :JoinProperty to identify that both properties were created by our approach and are part of the join integration.

Algorithm 8 Join integration

Input: G_I is the integrated graph, P_A and P_B are two data type properties, l is a user-provided label for the aligned resource, and S is a user-provided label for the join representation.
Output: Generates the integrated graph $G_{I'}$

```
function JoinIntegration(G_I, P_A, P_B, l, S)
    G_I' ← G_I
    A_unused' ← A_unused
    P_l ← generateURI(l)
    P_S ← generateURI(S)
    ...                                          ▹ Invariants I4, I3, I2
    ...                                          ▹ Start of Invariant I1
    G_I' ∪= ⟨P_l, rdf:type, :IntegratedResource⟩
    G_I' ∪= ⟨P_l, rdf:type, rdf:Property⟩
    G_I' ∪= ⟨P_A, rdfs:subPropertyOf, P_l⟩
    G_I' ∪= ⟨P_B, rdfs:subPropertyOf, P_l⟩
    C_rangeA ← GetIntegratedClass(G_I', P_A, rdfs:range)
    G_I' ∪= ⟨P_l, rdfs:range, C_rangeA⟩
    G_I' ∪= ⟨P_S, rdf:type, rdf:Property⟩
    G_I' ∪= ⟨P_l, :JoinProperty, P_S⟩
    ...                                          ▹ End of Invariant I1
    return G_I'
```

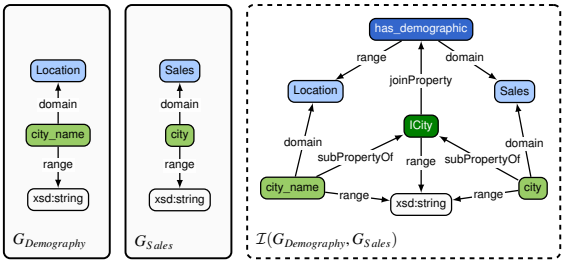

Fig. 12. Example of a Join integration

5.2. Incremental example

To perform an incremental integration, we will use the final result of Example 5. For this case, consider the data analyst wants to integrate the typed graph $\mathcal{I}(G_{CMOA}, G_{Cooperhewitt})$ in Figure 11 with the typed graph $G_{artists}$ in Figure 13. In this iteration, the schema matching approach proposed the alignments depicted in Table 5. First, we integrate rdfs:Class. Let us consider the alignment Creator and Artist. Note that we have two :IntegratedResource of type

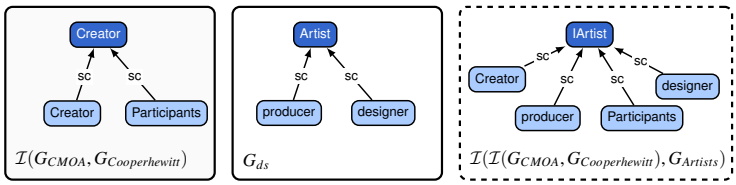

Fig. 14. Result of integrating two integrated classes from $\mathcal{I}(G_{CMOA}, G_{Cooperhewitt})$ and $G_{Artistsi}$

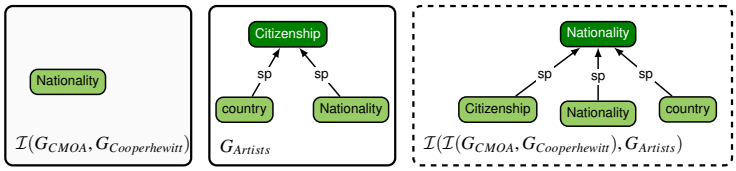

Fig. 15. Result of integrating one integrated data type property from $\mathcal{I}(G_{CMOA}, G_{Cooperhewitt})$ and $G_{Artists}$

class in this alignment resulting in the use of invariant $I4$ from Algorithm 2. Figure 14 illustrates the integration process for this alignment in which Creator and Artist are replaced by the new :IntegratedResource of type class, namely IArtist . Now, the algorithm proceeds to integrate Datatype property. Let us consider the alignment Nationality and Citizenship representing invariant $I2$. The integration process is illustrated in Figure 15. For this case, instead of creating a new :IntegratedResource of type property, we reuse Citizenship. Therefore, Nationality will be a rdfs:subPropertyOf Citizenship. For the last alignment, Name and Full_Name, we replace both :IntegratedResource of type property by Name. In summary, our approach creates one :IntegratedResource of type class and two :IntegratedResource of type property. The resulting integrated graph is depicted in Figure 16.

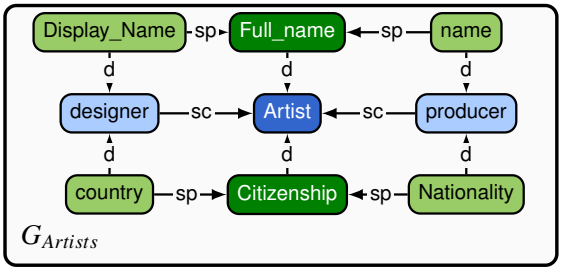

Fig. 13. New data source

$\mathcal{I}(G_{CMOA}, G_{Cooperhewitt})$	$G_{Artists}$	User-provided label l
Creator	Artist	Artist
Nationality	Citizenship	Nationality
Name	Full_name	Name

Table 5

Alignments discovered for $\mathcal{I}(G_{CMOA}, G_{Cooperhewitt})$ and $G_{Artists}$

5.3. Limitations

The integration algorithm presents an incremental approach to integrate typed graphs, however, data can be modeled in different ways which can in turn have an impact during the integration process. To illustrate this, consider the example presented in Figure 17. Let us consider, a data analyst interested to retrieve the country name related to a project. For graph G_{ds1}, this information can be obtained by using the property hasCountry to retrieve the Country entity. However, the graph G_{ds2} has a different way to represent the country information from a project requiring to retrieve Location entity using the property hasLocation and then using the property hasAddress to retrieve the PostalAddress entity which contains the country information. The main problem in this case is that the alignment is not resource to resource, instead the alignment involves a subgraph to one resource. As far as we know, there is no ontology alignment technique that considers this case and we for now do not either.

In this case, performing an integration between the graphs G_{ds1} and G_{ds2} using the resources Country - PostalAddress and name - country is not recommended. This will have an impact in the effort of integrating future sources as we have an evolving incremental graph where the entities and properties can be integrated only once into an integrated resource. Moreover, allowing integrations between entities with different granularities can cause unexpected and

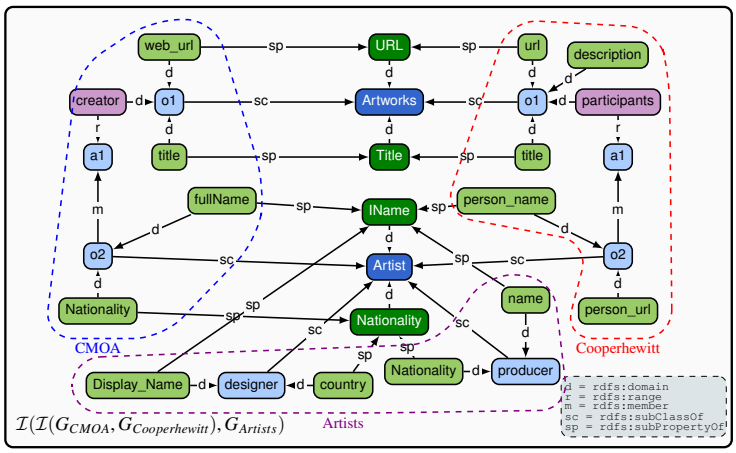

Fig. 16. Integration graph from a second integration

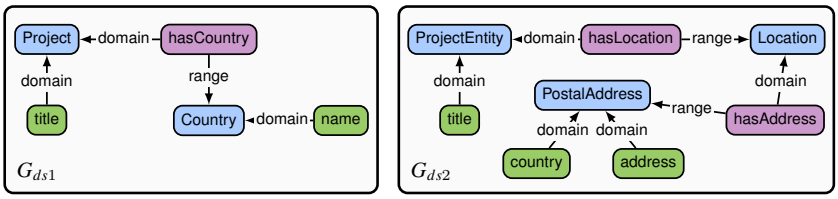

Fig. 17. Example of two different data models

unknown results when querying the sources. In this case, it is recommended to remodel the typed graph in the earliest integrations to the best modelization criteria according to the user needs. In summary, this work addresses only integration from resource to resource alignments, which is therefore affected by the modeling strategies in the sources. It is part of our future work to try to smooth this problem.

6. Evaluation

This section presents the evaluation results of the proposed approach which comprised a series of experiments to analyze runtime and complexity performance. To this end, we have implemented the NextiaDI library in ODIN [54], a DI system to virtually query heterogeneous data sources. Consequently, ODIN has a more efficient and complete end-to-end DI workflow by reducing the human effort in the generation of DI constructs, which were generated manually by users. With NextiaDI, we have efficiently generated source schemata for data sources, incrementally integrated schemas and derived DI constructs using the generated IG. As a query interface, ODIN presents the user with a global graph. Hence, we have generated a fully merged schema from the integration graph using Algorithm 9, which merges sub-classes and sub-properties into the integrated resource. As follows, we evaluate our two technical contributions (i.e., bootstrapping and schema integration) to assess their complexity and runtime performance. All details and reproducibility instructions can be found in the companion website[7]. All experiments were carried out on Intel core i5 having 16GB RAM and 2.3 GHz of the processor on Mac with Java compiler 11.

[7]https://www.essi.upc.edu/dtim/nextiadi/

Algorithm 9 Minimal integrated graph

Input: An integration graph G_I
Output: A fully merge integrated graph $G_{I'}$

```
function GENERATEMINIMALGRAPH(G_I)
    for all r in ⟨?R, rdf:type, :IntegrationResource⟩(G_I') do
        if ⟨r, rdf:type, rdf:Property⟩(G_I') then
            for all s in ⟨?s, rdfs:subPropertyOf, p⟩(G_I') do
                G_I' \= ⟨s, rdfs:domain, ?domain⟩(G_I')
                G_I' \= ⟨s, rdfs:range, ?range⟩(G_I')
                G_I' \= ⟨s, rdf:type, rdf:Property⟩
                G_I' \= ⟨s, rdfs:subPropertyOf, p⟩
        else if ⟨r, rdf:type, rdfs:Class⟩(G_I') then
            for all s in ⟨?s, rdfs:subClassOf, c⟩(G_I') do
                for all p in ⟨?p, rdfs:domain, s⟩(G_I') do
                    G_I' \= ⟨p, rdfs:domain, s⟩
                    G_I' ∪= ⟨p, rdfs:domain, c⟩
                for all p in ⟨?p, rdfs:range, s⟩(G_I') do
                    G_I' \= ⟨p, rdfs:range, s⟩
                    G_I' ∪= ⟨p, rdfs:range, c⟩
                G_I' \= ⟨s, rdf:type, rdfs:Class⟩
                G_I' \= ⟨s, rdfs:subClassOf, c⟩
    return G_I'
```

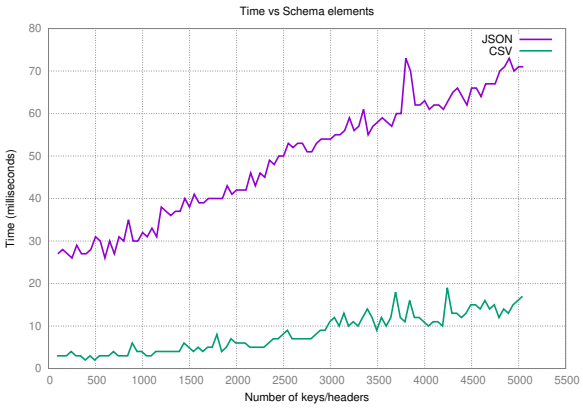

Fig. 18. Performance evaluation when the number of schema elements increased

6.1. Evaluation of Bootstrapping

We evaluated the bootstrapping of JSON and CSV data sources by measuring the impact of the schema size. Therefore, we increase the size of the schema elements. This experiment was executed 10 times. We describe the dataset preparation and the results obtained in the following.

Dataset preparation. We generated 100 datasets in JSON and CSV format. The initial JSON dataset contains a schema of 100 keys (9 objects and 91 attributes). We incrementally increased the number of schema elements by appending 50 new keys (1 object with 49 attributes) in the schema. For example, the second and third datasets will contain 150 keys (10 objects and 140 attributes) and 200 keys (11 objects and 189 attributes), respectively. For the initial CSV dataset, the schema contains 91 headers and we incrementally appended 49 new headers.

Results. Figure 18 depicts the correlation between time in milliseconds and the number of keys/headers. Note that in both data sources, the time to generate a typed graph schema depends on the size of the schema. JSON bootstrapping requires more time than CSV bootstrapping since the algorithm will parse one instance of the JSON to generate and homogenize the JSON schema and apply the production rules. The initial dataset took 27 milliseconds to produce a typed graph, while the last dataset, with a schema of 108 objects and 4942 attributes, took 71 milliseconds. In contrast, CSV bootstrapping requires only the header information to produce a typed graph schema. The initial dataset took 3 milliseconds to produce the typed graph, and the last dataset, with a schema of 4942 headers, took 17 milliseconds. Overall, we can observe some peeks in the trend. However, we consider these peaks are anomalies produced due to the performance of the java garbage collector since all results are constant within milliseconds.

6.2. Evaluation of schema integration

We evaluate the schema integration under three scenarios: i) increasing the number of alignments in an incremental integration and ii) integrate the same number of alignments but the number of elements in schemas increases and iii) perform integration using real data

6.2.1. Experiment 1 — increased alignments

Dataset preparation. For this experiment, the algorithm requires the generation of schemas and alignments that will be integrated incrementally. In the case of schemas, we reuse the typed graph generated by the initial JSON dataset generated in section 6.1 to generate 100 typed graph schemas. The resulting schema contains 9 classes, 90 data type properties and 1 object property. We generated the alignments for each typed graph increasing by one the number of alignments. Therefore, the first integration generates 1 alignment and the last iteration 100 alignments.

Results. Figure 19 depicts the correlation between time in milliseconds and the number of alignments. We can observe that the time to integrate schemas depends on the number of alignments. The time to integrate one alignment took two milliseconds, while the integration of 100 alignments took 112 milliseconds. Moreover, in iteration 13, the number of integrated classes converged. Then, the remaining iteration reuses all integrated classes by applying invariant I3 from Algorithm 2. For data type and object properties, they converged in iterations 35 and 4, respectively. Then, the remaining iteration did not create any integrated properties. In addition, the final integrated graph contains 1000 classes, 9 integrated classes, 9000 datatype properties, 90 integrated properties, 900 object properties, and 1 integrated object property. Overall, the algorithm efficiently integrates schemas incrementally.

6.2.2. *Experiment 2 — increased schema elements*

Dataset preparation. For this experiment, we generated 100 schemas using JSON datasets where the number of elements in the schema were increasing by adding 50 keys. For the generation of alignments, we produced 100 alignments in each iteration since the goal is to measure the impact of the schema size in the integration.

Results. Figure 20 depicts the correlation between time in milliseconds and the total number of resources (e.g., classes and properties) in the integrated graph. We can observe that the size of the integrated graph impact the time for integrating alignments. The final integrated graph contains 5950 classes, 9 integrated classes, 256500 datatype properties, 90 integrated datatype properties, 5850 object properties and 1 integrated object property. In total 268400 resources. We consider the impact in the time due to the size of the graph schema is on how Jena manage the graph in memory.

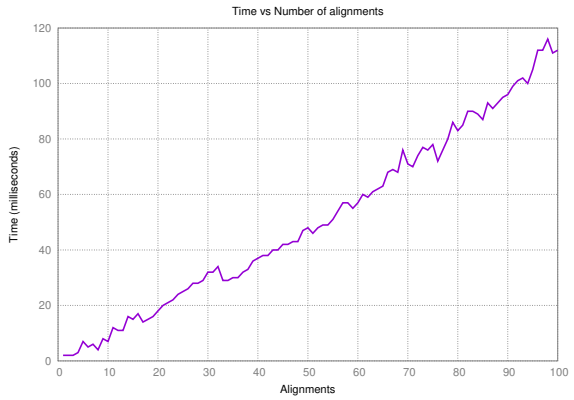

Fig. 19. Performance evaluation when alignments increased incrementally

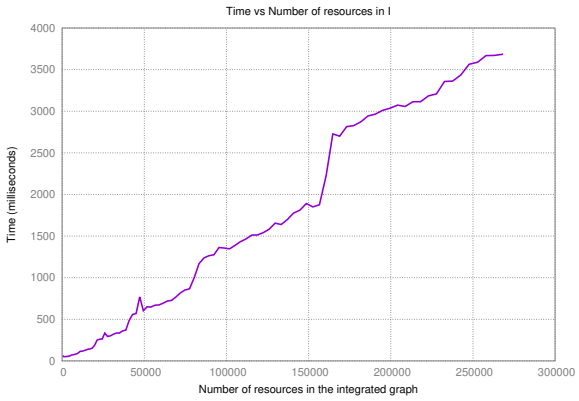

Fig. 20. Performance evaluation when the number of schema elements increased

6.2.3. *Experiment 3 — real data*

Dataset preparation. We have selected two tracks of the Ontology Alignment Evaluation Initiative: anatomy and large biomed track. The former contains the Foundational Model of Anatomy (FMA) with 2744 classes and part of the National Cancer Institute Thesaurus (NCI) with 3304 classes. The alignments provided are 1516 class alignments. The latter contains the FMA with 78998 classes and 15 properties, and NCI with 77269 classes and 186 properties. There are 2686 class alginments for this track.

Results. This integration was performed in one step. For the anatomy track all elements were integrated in 120 milliseconds. The final integrated graph contains 6048 classes and 1516 integrated classes. For the large biomed track all elements were integrated in 3043 milliseconds with 156267 classes, 2686 integrated classes, 78 data type properties and 123 object properties. Overall, our integration approach is efficient when dealing with large schemas as the biomed track.

7. Conclusions and future work

This paper presents an approach for efficiently generating schemas of heterogeneous data sources and incrementally integrating them to facilitate the end-to-end DI workflow. Our bootstrapping is generalizable to any data model and not conditional on the availability of predefined schemas, which are challenging to maintain when new data sources appear, as discussed in the related work section. Our integration approach proposes four main invariants to incrementally generate and propagate corresponding metadata in an integrated graph representing how resources of source schemas are integrated (e.g., union and join). We present the formal foundation for integrating classes and properties, though our approach can be extended by applying the proposed invariants to any resource. Finally, we encapsulate our approach into an open-source library, namely NextiaDI, to provide reusability and easy integration with DI ecosystem tools. We showcase the usability and effectiveness of NextiaDI in the ODIN system to automatically generate the required DI constructs. Our future work considers possible strategies for complex integrations as explained in Section 5.3 to detect and automatize as much as possible those cases as well as enhance other phases of the end-to-end DI workflow, such as schema matching.

Acknowledgements. This work was partly supported by the DOGO4ML project, funded by the Spanish Ministerio de Ciencia e Innovación under project PID2020-117191RB-I00. Javier Flores is supported by contract 2020-DI-027 of the Industrial Doctorate Program of the Government of Catalonia and Consejo Nacional de Ciencia y Tecnología (CONACYT, Mexico). Sergi Nadal is partly supported by the Spanish Ministerio de Ciencia e Innovación, as well as the European Union - NextGenerationEU, under project FJC2020-045809-I.

References

[1] R.J. Miller, Open Data Integration, *Proc. VLDB Endow.* **11**(12) (2018), 2130–2139.
[2] R. Kitchin, *The data revolution: Big data, open data, data infrastructures and their consequences*, Sage, 2014.
[3] H. Song, R. Srinivasan, T. Sookoor and S. Jeschke, *Smart cities: foundations, principles, and applications*, John Wiley & Sons, 2017.
[4] B. Golshan, A.Y. Halevy, G.A. Mihaila and W. Tan, Data Integration: After the Teenage Years, in: *PODS*, ACM, 2017, pp. 101–106.
[5] M. Chen, S. Mao and Y. Liu, Big Data: A Survey, *Mob. Networks Appl.* **19**(2) (2014), 171–209.
[6] S. Bergamaschi, D. Beneventano, A. Corni, E. Kazazi, M. Orsini, L. Po and S. Sorrentino, The Open Source release of the MOMIS Data Integration System, in: *SEBD*, 2011, pp. 175–186.
[7] S.S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J.D. Ullman and J. Widom, The TSIMMIS Project: Integration of Heterogeneous Information Sources, in: *IPSJ*, 1994, pp. 7–18.
[8] M. Solanki, C. Mader, H. Nagy, M. Mückstein, M. Hanfi, R. David and A. Koller, Ontology-Driven Unified Governance in Software Engineering: The PoolParty Case Study, in: *ESWC (2)*, Lecture Notes in Computer Science, Vol. 10250, 2017, pp. 109–124.
[9] C.A. Knoblock, P.A. Szekely, J.L. Ambite, A. Goel, S. Gupta, K. Lerman, M. Muslea, M. Taheriyan and P. Mallick, Semi-automatically Mapping Structured Sources into the Semantic Web, in: *ESWC*, Lecture Notes in Computer Science, Vol. 7295, Springer, 2012.
[10] S.J.R. Méndez, A. Haller, P.G. Omran, J. Wright and K. Taylor, J2RM: An ontology-based JSON-to-RDF Mapping tool, in: *ISWC (Demos/Industry)*, CEUR Workshop Proceedings, Vol. 2721, CEUR-WS.org, 2020, pp. 368–373.
[11] M. Hacherouf, S.N. Bahloul and C. Cruz, Transforming XML documents to OWL ontologies: A survey, *J. Inf. Sci.* **41**(2) (2015), 242–259.
[12] P. Ziegler and K.R. Dittrich, *Data Integration — Problems, Approaches, and Perspectives*, in: *Conceptual Modelling in Information Systems Engineering*, J. Krogstie, A.L. Opdahl and S. Brinkkemper, eds, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 39–58. ISBN 978-3-540-72677-7. doi:10.1007/978-3-540-72677-7_3. https://doi.org/10.1007/978-3-540-72677-7_3.
[13] G. Flouris, D. Manakanatas, H. Kondylakis, D. Plexousakis and G. Antoniou, Ontology change: classification and survey, *Knowl. Eng. Rev.* **23**(2) (2008), 117–152.
[14] H. Kondylakis, G. Flouris and D. Plexousakis, Ontology and Schema Evolution in Data Integration: Review and Assessment, in: *OTM Conferences (2)*, Lecture Notes in Computer Science, Vol. 5871, Springer, 2009, pp. 932–947.
[15] B.E. Idrissi, S. Baïna and K. Baïna, Automatic generation of ontology from data models: A practical evaluation of existing approaches, in: *RCIS*, IEEE, 2013, pp. 1–12.
[16] M. Stonebraker and I.F. Ilyas, Data Integration: The Current Status and the Way Forward, *IEEE Data Eng. Bull.* **41**(2) (2018), 3–9.
[17] S. Raunich and E. Rahm, Target-driven merging of taxonomies with Atom, *Inf. Syst.* **42** (2014), 1–14.
[18] S. Amrouch and S. Mostefai, Survey on the literature of ontology mapping, alignment and merging, in: *2012 International Conference on Information Technology and e-Services*, 2012, pp. 1–5.
[19] I. Osman, S.B. Yahia and G. Diallo, Ontology Integration: Approaches and Challenging Issues, *Inf. Fusion* **71** (2021), 38–63.
[20] N.W. Paton, K. Belhajjame, S.M. Embury, A.A.A. Fernandes and R. Maskat, Pay-as-you-go Data Integration: Experiences and Recurring Themes, in: *SOFSEM*, Lecture Notes in Computer Science, Vol. 9587, Springer, 2016, pp. 81–92.

[21] L.M. Haas, Beauty and the Beast: The Theory and Practice of Information Integration, in: *ICDT*, Lecture Notes in Computer Science, Vol. 4353, Springer, 2007, pp. 28–43.
[22] Ó. Corcho, F. Priyatna and D. Chaves-Fraga, Towards a new generation of ontology based data access, *Semantic Web* **11**(1) (2020).
[23] P. Atzeni, P. Cappellari, R. Torlone, P.A. Bernstein and G. Gianforme, Model-independent schema translation, *VLDB J.* **17**(6) (2008).
[24] C. Pinkel, C. Binnig, E. Jiménez-Ruiz, E. Kharlamov, A. Nikolov, A. Schwarte, C. Heupel and T. Kraska, IncMap: A Journey towards Ontology-based Data Integration, in: *BTW*, LNI, Vol. P-265, GI, 2017, pp. 145–164.
[25] C. Bizer and R. Cyganiak, D2r server-publishing relational databases on the semantic web, in: *Poster at the 5th international semantic web conference*, Vol. 175, 2006.
[26] E. Jiménez-Ruiz, E. Kharlamov, D. Zheleznyakov, I. Horrocks, C. Pinkel, M.G. Skjæveland, E. Thorstensen and J. Mora, BootOX: Practical Mapping of RDBs to OWL 2, in: *ISWC (2)*, Lecture Notes in Computer Science, Vol. 9367, Springer, 2015, pp. 113–132.
[27] L.F. de Medeiros, F. Priyatna and Ó. Corcho, MIRROR: Automatic R2RML Mapping Generation from Relational Databases, in: *ICWE*, Lecture Notes in Computer Science, Vol. 9114, Springer, 2015, pp. 326–343.
[28] I. Bedini, C.J. Matheus, P.F. Patel-Schneider, A. Boran and B. Nguyen, Transforming XML Schema to OWL Using Patterns, in: *ICSC*, IEEE Computer Society, 2011, pp. 102–109.
[29] C. Tsinaraki and S. Christodoulakis, XS2OWL: A Formal Model and a System for Enabling XML Schema Applications to Interoperate with OWL-DL Domain Knowledge and Semantic Web Tools, in: *DELOS*, Lecture Notes in Computer Science, Vol. 4877, Springer, 2007.
[30] P.T.T. Thuy, Y. Lee and S. Lee, DTD2OWL: automatic transforming XML documents into OWL ontology, in: *ICIS*, ACM International Conference Proceeding Series, Vol. 403, ACM, 2009, pp. 125–131.
[31] K.M. Albarrak and E.H. Sibley, A survey of methods that transform data models into Ontology models, in: *IRI*, IEEE Systems, Man, and Cybernetics Society, 2011, pp. 58–65.
[32] J.F. Sequeda, S.H. Tirmizi, Ó. Corcho and D.P. Miranker, Survey of directly mapping SQL databases to the Semantic Web, *Knowl. Eng. Rev.* **26**(4) (2011), 445–486.
[33] J. Flores, S. Nadal and O. Romero, Towards Scalable Data Discovery, in: *EDBT*, OpenProceedings.org, 2021, pp. 433–438.
[34] G. Xiao, L. Ding, B. Cogrel and D. Calvanese, Virtual Knowledge Graphs: An Overview of Systems and Use Cases, *Data Intell.* **1**(3) (2019), 201–223.
[35] N.F. Noy and M.A. Musen, PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment, in: *AAAI/IAAI*, AAAI Press / The MIT Press, 2000, pp. 450–455.
[36] D.L. McGuinness, R. Fikes, J. Rice and S. Wilder, The Chimaera Ontology Environment, in: *AAAI/IAAI*, AAAI Press / The MIT Press, 2000, pp. 1123–1124.
[37] G. Stumme and A. Maedche, FCA-MERGE: Bottom-Up Merging of Ontologies, in: *IJCAI*, Morgan Kaufmann, 2001, pp. 225–234.
[38] P. Mitra, G. Wiederhold and S. Decker, A Scalable Framework for the Interoperation of Information Sources, in: *SWWS*, 2001, pp. 317–329.
[39] D. Dou, D.V. McDermott and P. Qi, Ontology Translation on the Semantic Web, *J. Data Semant.* **2** (2005), 35–57.
[40] S. Babalou and B. König-Ries, Towards Building Knowledge by Merging Multiple Ontologies with CoMerger: A Partitioning-based Approach, *CoRR* **abs/2005.02659** (2020).
[41] B. Vidé, J. Marty, F. Ravat and M. Chevalier, Designing a Business View of Enterprise Data: An approach based on a Decentralised Enterprise Knowledge Graph, in: *IDEAS*, ACM, 2021, pp. 184–193.
[42] S. Babalou, E. Grygorova and B. König-Ries, What to Do When the Users of an Ontology Merging System Want the Impossible? Towards Determining Compatibility of Generic Merge Requirements, in: *EKAW*, Lecture Notes in Computer Science, Vol. 12387, Springer, 2020.
[43] M. Giese, A. Soylu, G. Vega-Gorgojo, A. Waaler, P. Haase, E. Jiménez-Ruiz, D. Lanti, M. Rezk, G. Xiao, Ö.L. Özçep and R. Rosati, Optique: Zooming in on Big Data, *Computer* **48**(3) (2015), 60–67.
[44] C. Civili, M. Console, G.D. Giacomo, D. Lembo, M. Lenzerini, L. Lepore, R. Mancini, A. Poggi, R. Rosati, M. Ruzzi, V. Santarelli and D.F. Savo, MASTRO STUDIO: Managing Ontology-Based Data Access applications, *Proc. VLDB Endow.* **6**(12) (2013), 1314–1317.
[45] M.N. Mami, D. Graux, S. Scerri, H. Jabeen, S. Auer and J. Lehmann, Squerall: Virtual Ontology-Based Access to Heterogeneous and Large Data Sources, in: *ISWC (2)*, Lecture Notes in Computer Science, Vol. 11779, Springer, 2019, pp. 229–245.
[46] E. Kharlamov, S. Brandt, M. Giese, E. Jiménez-Ruiz, S. Lamparter, C. Neuenstadt, Ö.L. Özçep, C. Pinkel, A. Soylu, D. Zheleznyakov, M. Roshchin, S. Watson and I. Horrocks, Semantic Access to Siemens Streaming Data: the Optique Way, in: *ISWC (Posters & Demos)*, CEUR Workshop Proceedings, Vol. 1486, CEUR-WS.org, 2015.
[47] D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro and G. Xiao, Ontop: Answering SPARQL queries over relational databases, *Semantic Web* **8**(3) (2017), 471–487.
[48] D. Calvanese, G.D. Giacomo, M. Lenzerini and M.Y. Vardi, Simplifying schema mappings, in: *ICDT*, ACM, 2011, pp. 114–125.
[49] M.N. Mami, D. Graux, S. Scerri, H. Jabeen and S. Auer, Querying Data Lakes using Spark and Presto, in: *WWW*, ACM, 2019.
[50] A. Bonifati, G.H.L. Fletcher, H. Voigt and N. Yakovets, *Querying Graphs*, Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2018.
[51] M. Lenzerini, Data Integration: A Theoretical Perspective, in: *PODS*, ACM, 2002, pp. 233–246.
[52] J.L.C. Izquierdo and J. Cabot, Discovering Implicit Schemas in JSON Data, in: *ICWE*, Lecture Notes in Computer Science, Vol. 7977, Springer, 2013, pp. 68–83.
[53] E. Jiménez-Ruiz and B.C. Grau, LogMap: Logic-Based and Scalable Ontology Matching, in: *ISWC (1)*, Lecture Notes in Computer Science, Vol. 7031, Springer, 2011, pp. 273–288.
[54] S. Nadal, K. Rabbani, O. Romero and S. Tadesse, ODIN: A Dataspace Management System, in: *ISWC (Satellites)*, CEUR Workshop Proceedings, Vol. 2456, CEUR-WS.org, 2019, pp. 185–188.

Appendix A. JSON metamodel constraints

In this appendix, we present the constraints considered for the metamodel we adopt to represent the schemata of JSON datasets (i.e., $\mathcal{M}_{\text{JSON}}$), which is depicted in Section 4.1.1. Hereinafter, we assume all constraints are applied over a graph G which is typed with respect to $\mathcal{M}_{\text{JSON}}$.

IS-A relationships and constraints on generalizations. Rule 1 restricts instances of J:DataType to be instances of either J:Primitive, J:Object or J:Array. Then, Rules 2, 3, and 4 constrain that for any of such subclass instances, the instantiation of the superclass J:DataType also exists in G. Finally, Rules 5, 6 and 7 determine that the subclasses of J:DataType are disjoint.

$$\forall d\big(\langle d, \texttt{rdf:type}, \texttt{J:DataType}\rangle(G)\big) \Rightarrow \exists d'\big(\langle d', \texttt{rdf:type}, \texttt{J:Primitive}\rangle \vee \langle d', \texttt{rdf:type}, \texttt{J:Object}\rangle \vee \langle d', \texttt{rdf:type}, \texttt{J:Array}\rangle(G)\big) | d = d' \tag{1}$$

$$\forall o\big(\langle o, \texttt{rdf:type}, \texttt{J:Object}\rangle(G)\big) \Rightarrow \exists d\big(\langle d, \texttt{rdf:type}, \texttt{J:DataType}\rangle(G)\big) | o = d \tag{2}$$

$$\forall p\big(\langle p, \texttt{rdf:type}, \texttt{J:Primitive}\rangle(G)\big) \Rightarrow \exists d\big(\langle d, \texttt{rdf:type}, \texttt{J:DataType}\rangle(G)\big) | p = d \tag{3}$$

$$\forall a\big(\langle a, \texttt{rdf:type}, \texttt{J:Array}\rangle(G)\big) \Rightarrow \exists d\big(\langle d, \texttt{rdf:type}, \texttt{J:DataType}\rangle(G)\big) | a = d \tag{4}$$

$$\forall o\big(\langle o, \texttt{rdf:type}, \texttt{J:Object}\rangle(G)\big) \Rightarrow \nexists p\big(\langle p, \texttt{rdf:type}, \texttt{J:Primitive}\rangle(G)\big) | o = p \tag{5}$$

$$\forall o\big(\langle o, \texttt{rdf:type}, \texttt{J:Object}\rangle(G)\big) \Rightarrow \nexists a\big(\langle a, \texttt{rdf:type}, \texttt{J:Array}\rangle(G)\big) | o = a \tag{6}$$

$$\forall p\big(\langle p, \texttt{rdf:type}, \texttt{J:Primitive}\rangle(G)\big) \Rightarrow \nexists a\big(\langle a, \texttt{rdf:type}, \texttt{J:Array}\rangle(G)\big) | p = a \tag{7}$$

Referential integrity constraints. Rule 8 indicates that an edge labeled with J:hasValue will connect instances of either J:Document and J:Object, or instances of J:Key and J:DataType. Similarly, Rule 9 applies the same strategy to connect instances of J:Object and J:Key using edges labeled J:hasKey. Finally, for the J:hasMember edge label, we use it to connect instances of J:Array to instances of J:DataType. This property is expressed in Rule 10.

$$\forall d, o\big(\langle d, \texttt{J:hasValue}, o\rangle(G)\big) \Rightarrow \exists d', o'\Big(\big(\langle d', \texttt{rdf:type}, \texttt{J:Document}\rangle(G) \wedge \langle o', \texttt{rdf:type}, \texttt{J:Object}\rangle(G)\big) \vee \big(\langle d', \texttt{rdf:type}, \texttt{J:Key}\rangle(G) \wedge \langle o', \texttt{rdf:type}, \texttt{J:DataType}\rangle(G)\big)\Big) | d = d' \wedge o = o' \tag{8}$$

$$\forall o, k\big(\langle o, \texttt{J:hasKey}, k\rangle(G)\big) \Rightarrow \exists o', k'\big(\langle o', \texttt{rdf:type}, \texttt{J:Object}\rangle \wedge \langle k', \texttt{rdf:type}, \texttt{J:Key}\rangle(G)\big) | o = o' \wedge k = k' \tag{9}$$

$$\forall a, d\big(\langle a, \texttt{J:hasMember}, d\rangle(G)\big) \Rightarrow \exists a', d'\big(\langle a', \texttt{rdf:type}, \texttt{J:Array}\rangle \wedge \langle d', \texttt{rdf:type}, \texttt{J:DataType}\rangle(G)\big) | a = a' \wedge d = d' \tag{10}$$

Cardinality constraints. Rule 11 states that an instance of J:Document has a single instance of J:Object as root. Then, Rule 12 models a many-to-one relationship between instances of J:Key and J:DataType.

$$\forall d, o\big(\langle d, \texttt{J:hasValue}, o\rangle(G)\big) \Rightarrow \nexists d', o', o''\big(\langle d', \texttt{J:hasValue}, o'\rangle(G) \wedge \langle d', \texttt{J:hasValue}, o''\rangle(G)\big) | d = d' \wedge o = o' \wedge o = o'' \tag{11}$$

$$\forall k, d\big(\langle k, \texttt{J:hasValue}, d\rangle(G)\big) \Rightarrow \nexists k', d'\big(\langle k', \texttt{J:hasValue}, d'\rangle(G) | k = k' \wedge d = d' \tag{12}$$

Appendix B. RDFS metamodel constraints

In this appendix, we present the constraints considered for the fragment of RDFS that we consider in this paper (i.e., $\mathcal{M}_{\text{RDFS}}$), which is depicted in Section 4.1.2. All such constraints are based on the *RDF Schema 1.1* standard[8]. Hereinafter, we assume all constraints are applied over a graph G which is typed with respect to $\mathcal{M}_{\text{RDFS}}$.

IS-A relationships and constraints on generalizations. Rule 13 restricts instances of rdfs:Resource to be instances of either rdf:Property, rdfs:Class or Container. Then, Rules 14, 15, and 16 constrain that for any of such subclass instances, the instantiation of the superclass rdfs:Resource also exists in G. Next, Rules 17 and 18 denote that, respectively, instances of rdfs:Datatype are also instances of rdfs:Class and instances of rdf:Seq are also instances of rdfs:Container.

$$\forall r\big(\langle r, \texttt{rdf:type}, \texttt{rdfs:Resource}\rangle(G)\big) \Rightarrow \exists r'\big(\langle r', \texttt{rdf:type}, \texttt{rdf:Property}\rangle(G) \vee \langle r', \texttt{rdf:type}, \texttt{rdfs:Class}\rangle(G) \vee \langle r', \texttt{rdf:type}, \texttt{rdfs:Container}\rangle(G)\big) | r = r' \tag{13}$$

$$\forall p\big(\langle p, \texttt{rdf:type}, \texttt{rdf:Property}\rangle(G)\big) \Rightarrow \exists r\big(\langle r, \texttt{rdf:type}, \texttt{rdfs:Resource}\rangle(G)\big) | p = r \tag{14}$$

$$\forall c\big(\langle c, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle(G)\big) \Rightarrow \exists r\big(\langle r, \texttt{rdf:type}, \texttt{rdfs:Resource}\rangle(G)\big) | c = r \tag{15}$$

$$\forall c\big(\langle c, \texttt{rdf:type}, \texttt{rdfs:Container}\rangle(G)\big) \Rightarrow \exists r\big(\langle r, \texttt{rdf:type}, \texttt{rdfs:Resource}\rangle(G)\big) | c = r \tag{16}$$

$$\forall d\big(\langle d, \texttt{rdf:type}, \texttt{rdfs:Datatype}\rangle(G)\big) \Rightarrow \exists c\big(\langle c, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle(G)\big) | d = c \tag{17}$$

$$\forall s\big(\langle s, \texttt{rdf:type}, \texttt{rdf:Seq}\rangle(G)\big) \Rightarrow \exists c\big(\langle c, \texttt{rdf:type}, \texttt{rdfs:Container}\rangle(G)\big) | s = c \tag{18}$$

Referential integrity constraints. Rule 19 indicates that the rdfs:domain of an instance of rdf:Property is an instance of rdfs:Class. Similarly, Rule 20, constraints that the rdfs:range of instances of rdf:Property are also instances of rdfs:Class.

$$\forall p, c\big(\langle p, \texttt{rdfs:domain}, c\rangle(G)\big) \Rightarrow \exists p', c'\big(\langle p', \texttt{rdf:type}, \texttt{rdf:Property}\rangle(G) \wedge \langle c', \texttt{rdf:type}, \texttt{rdfs:Class}\rangle(G)\big) | p = p' \wedge c = c' \tag{19}$$

$$\forall p, c\big(\langle p, \texttt{rdfs:range}, c\rangle(G)\big) \Rightarrow \exists p', c'\big(\langle p', \texttt{rdf:type}, \texttt{rdf:Property}\rangle(G) \wedge \langle c', \texttt{rdf:type}, \texttt{rdfs:Class}\rangle(G)\big) | p = p' \wedge c = c' \tag{20}$$

B.1. Schema integration constraints

Here we present the constraints for the RDFS metamodel extension we consider to support the annotated integration process depicted in Section 5.

IS-A relationships. Rule 21 states that any instance of :IntegratedResource is also an instance of rdfs:Resource. Similarly, Rule 22, denotes that any instance of :JoinProperty is also an instance of rdf:Property.

$$\forall p\big(\langle p, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle(G)\big) \Rightarrow \exists r\big(\langle r, \texttt{rdf:type}, \texttt{rdfs:Resource}\rangle(G)\big) | p = r \tag{21}$$

$$\forall p\big(\langle p, \texttt{rdf:type}, \texttt{:JoinProperty}\rangle(G)\big) \Rightarrow \exists r\big(\langle r, \texttt{rdf:type}, \texttt{rdf:Property}\rangle(G)\big) | p = r \tag{22}$$

Referential integrity constraints. Rule 23 denotes that the rdfs:domain of a :JoinProperty instance is an :IntegratedResource.

$$\forall p, c\big(\langle p, \texttt{rdfs:domain}, c\rangle(G)\big) \Rightarrow \exists p', c'\big(\langle p', \texttt{rdf:type}, \texttt{:JoinProperty}\rangle(G) \wedge \langle c', \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle(G)\big) | p = p' \wedge c = c' \tag{23}$$

[8] https://www.w3.org/TR/rdf-schema/